# EFFICIENT, EXPRESSIVE, AND EFFECTIVE RUNTIME VERIFICATION

BY

PATRICK O'NEIL MEREDITH

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Associate Professor Grigore Roşu, Chair and Director of Research
Senior Research Scientist Klaus Havelund, JPL Laboratory for Reliable Software
Associate Professor Marco Caccamo
Associate Professor Darko Marinov

# Abstract

Runtime Verification is a quickly growing technique for providing many of the guarantees of formal verification, but in a manner that is scalable. It useful information available from actual runs of programs to make verification decisions, rather than the purely static information used in formal verification.

One of the main facets of Runtime Verification is runtime monitoring, where safety properties are checked against the execution of a program during (or in some cases after) its run. Prior work on efficient monitoring focused primarily on finite state properties. Non-finite state techniques existed, but added orders of magnitude of runtime overhead on the monitored system. The vast majority of runtime monitoring has also been limited to the application domain, with violations of safety properties only found on the actual trace of a given program.

This thesis describes research that demonstrates that various logical formalisms, including those more powerful than finite logics, can be efficiently monitored in multiple monitoring domains. The demonstrated monitoring domains run the gamut from the application level with the Java programming language, to monitoring traces *predicted* from a given run of a program, to hardware based monitors designed to ensure proper peripheral operation. The logical formalisms include multicategory finite state machines, extended regular expressions, past-time linear temporal logic with optimization for hardware based monitors, context-free grammars, linear temporal logic with both past and future operators, and string rewriting. This combination of domains and logical formalisms show that monitoring can be both expressive and efficient, regardless of the expressive power of the logical formalism, and that monitoring can be used not only for flat traces generated by software applications, but also in predictive traces and a hardware context.

# Acknowledgments

First, I must thank my parents for their constant support, both emotionally and monetarily. Without them, this would not be possible.

I would like to thank my committee for having the patience to read this monolith, and more specifically: I would like to thank Klaus for listening to my random musings on gchat, Marco for his work on BusMOP, Darko for being a good sport, and Grigore... for pretty much everything; it's been an interesting ride.

Dongyun Jin and Feng Chen, may he rest in peace, have both been especially instrumental to my work and have been good friends; I cannot thank them enough.

To my friends, for their necessary distractions keeping me sane over the years: Aaron Hosek, Andre Ştefănescu, Andrew Belitto, Andrew Lenharth, Angela Sze, Barbara Bibiloni Clua, Brandon Cottrell, Brandon Rollins, Brian Urbanek, Carrie Heyen, Chucky Ellison, Chad Zalkin, Cody Slauson, Dan Watson, Dan Winkle, Deepak Ramachandran, Dennis Griffith, Jared McCloy, Jason Evenden, Jay Gentile, Josh Holman, Joe Tucek, Josh Laughlin, Mark Hills, Michael Katelman, Michael Ilseman, Michael Matsche, Meghan Jones, Nelson Parrot, Nic Miller, Nitish Korula, Oksana Tkachuk, Ralf Sasse, Steve Lauterburg, Rob Bocchino, Traian Şerbănuţă, and Zac Cochran. Extra special mention to Joseph Jackan for "feats of unrestrained manliness".

Thanks to Margaret Pennell for always believing in me.

Thanks to all the McCalls and Merediths.

Thanks to Metal Music, particularly Black, for keeping my focus. Especially thanks to Ulvhedin Hoest.

Thanks to the St. Louis Cardinals for winning two World Series while I was in grad school.

The research in this dissertation has been supported in part by NSF grants CCF-0916893, CNS-0720512, CNS-0509321, CCF-0448501, by NASA contract NNL08AA23C, by a Samsung SAIT grant, and by several Microsoft gifts.

# Table of Contents

# Chapter 1

# Introduction

## 1.1 Problem Description

Computers have become an increasingly large part of our lives. Nearly everything we use now contains computers: they control the fuel injection in our cars, they run our televisions, and our phones are now computing devices. Computers also also relied upon for safety critical systems such as flight controls in passenger aircraft. All of these computers are controlled by increasingly complex software. This ever growing reliance on software requires new methods to improve software reliability.

Runtime monitoring of requirements, most often specified as safety policies, can increase the reliability of the resulting hardware or software systems. There is an increasingly broad interest in uses of monitoring in software development and analysis, as reflected, for example, by abundant approaches proposed recently ([6, 17, 23, 29, 41, 44, 54, 58, 81, 89] among others), and also by the Runtime Verification (RV) and the formal aspects of testing (FATES) initiatives [15, 57, 58, 60, 61, 117] among many others. Many of these approaches support what is know as *parametric* monitoring, wherein different instances of objects in a system can be bound to parameters. Parametric properties are a generalization of the well-known concept of typestates [118] that allow for specifying safety properties that describe the *interactions between objects*, rather than properties of single objects.[1] Most of these techniques, however, have either been focused on finite-state properties or have added orders of magnitude overhead to the monitored systems. Whatever the power of the formalism used, all of the mentioned systems fix the choice of formalism. This thesis shows that many possible logical formalisms can be used, including those more powerful than finite state, while still maintaining efficiency. This goal is achieved through a high efficiency formalism-agnostic algorithm for parametric monitoring, as well as

[1] In this respect, typestates can be thought of as parametric properties with one parameter.

very efficient monitoring algorithms for the formalisms in question. The software runtime monitoring presented in this thesis is embodied within the Monitoring Oriented Programming (MOP) Framework, specifically JavaMOP, the Java instance of the MOP framework.

Hardware approaches to monitoring have seen less active research. Most attempts have been for the purpose of performance measurement or temperature control rather than the safety properties with which we are concerned. [88] is an approach that generates monitors from formal safety properties that are implemented in hardware, but these hardware monitors are actually used to monitor software programs, making it fit more with the software approaches mentioned above than the hardware based monitoring of the BusMOP developed for this thesis. BusMOP is an instance of the MOP framework for monitoring bus traffic, which was developed to to satisfy the goal of monitoring formal safety properties with hardware monitors, and is presented in this thesis.

Predictive runtime analysis [34, 37], itself a form of Runtime Verification, is able to run a program, collect logs, and reconstruct a causal model of the program that can be used to infer *possible* executions of a programs. These inferred executions can be compared against the formal properties designed for monitoring software applications. The RV-Predict system, developed for this thesis, shows that predictive analysis, which was previously applied only to race detection and atomicity checking, can be applied to formally stated properties written in any of the logical formalisms described here. It also improves the efficiency of predictive runtime analysis by orders of magnitude, making it applicable to a larger set of programs.

The performance and expressivity advances in the realm of software monitoring presented in this thesis and embodied in the JavaMOP instance of the MOP framework work to further the goals of this thesis: efficient, expressive, and effective Runtime Verification. The hardware monitoring of BusMOP and the predictive monitoring of RV-Predict show that Runtime Verification can be effective in totally new domains.

**Contributions.** The contributions of this thesis can be broken into eight major areas. The individual areas well be presented in-depth in the remainder of this thesis.

- The design of the MOP framework, which currently contains two instances: JavaMOP, BusMOP. See Chapter 2.

- Formalism-generic software monitoring performance improvements. Two optimizations revolving around static analysis of the *property* to be monitored are presented. They both are formalism independent and lead to one of the most efficient software monitoring systems in the world. See Chapter 3.

- Formalism-generic software monitoring expressivity improvements. Formalism generic suffix monitoring and parameter binding modes allow for more expressive control of monitoring, regardless of logical formalism used. See Chapter 3.

- Formalism-generic hardware bus monitoring implementation. BusMOP shows that Runtime Verification is both an effective and efficient means to provide guarantees in hardware and hardware/software hybrid systems. See Chapter 4.

- Efficient algorithms for monitoring finite formalisms. Several algorithms are presenting for converting all of our finite logical formalisms into finite state machines. This is important both because of an algorithm for minimizing multicategory finite state machines, as well as reducing the amount of static analysis algorithms needed to allow for the optimizations presented in Chapter 3. See Chapter 5.

- An efficient algorithm for monitoring context-free properties. An in depth discussion of context-free grammars used for monitoring is presented, as well as an efficient algorithm with a proof of correctness. This is the first truly efficient means for monitoring parametric context-free grammars. See Chapter 6.

- An efficient algorithm for monitoring string rewriting systems. An algorithm for efficiently monitoring parametric string rewriting systems is shown. As string rewriting systems are Turing-complete, we provide the first implementation of an efficient Turing-complete logical formalism monitoring algorithm. See Chapter 7.

- Formalism-generic software predictive analysis implementation. We present the first system to causally predict generic safety properties within a cone of possible thread interleavings. Several theoretical and engineering decisions are made to improve performance. This demonstrates the efficacy of Runtime Verification in a new domain. See Chapter 8.

Additionally, Chapter 3 features an in-depth formal treatment of the semantics of monitoring.

## 1.2 Summary of Related Work

What follows is a summary of related work so that there may be one point to refer to. The related work presented here is aimed to give an overview of the work most directly related to the topics of this thesis. Additional, more in-depth related work is available in the respective chapters, where appropriate.

First, we discuss the relationships between the MOP framework and other related paradigms, including AOP, design by contract, Runtime Verification, and other trace monitoring approaches. Broadly speaking, all the approaches discussed below are instances of runtime monitoring. Interestingly, even though most of the systems mentioned below target the same programming languages, only two share the exact same logical formalism for expressing properties. This observation strengthens our belief that probably there is *no silver bullet logic* (or *super logic*) for all purposes. A major objective in the design of the MOP framework was to avoid hardwiring particular logical formalisms into the system.

### 1.2.1 Aspect Oriented Programming (AOP) Languages

Since its proposal in [80], AOP has been increasingly adopted and many tools have been developed to support AOP in different programming languages, e.g., AspectJ and JBoss [75] for Java, and AspectC++ [7] for C++. Built on these general AOP languages, numerous extensions have been proposed to provide domain-specific features for AOP. Among these extensions, Tracematches [6] and J-LO [23] support history(trace)-based aspects for Java.

Tracematches enables the programmer to trigger the execution of certain code by specifying a parametric regular pattern of events in a computation trace, where the events are defined over entry/exit of AspectJ pointcuts. When the pattern is matched during the execution, the associated code will be executed. In this sense, Tracematches supports trace-based pointcuts for AspectJ. J-LO is a tool for runtime-checking temporal assertions. These temporal assertions are specified using parametric linear temporal logic (LTL) and the syntax adopted in J-LO is similar to Tracematches' except that the properties are specified in a different formalism. J-LO also uses the same parametricity semantics as Tracematches.

J-LO mainly focuses on checking at runtime properties rather than providing programming support. In J-LO, the temporal assertions are inserted into Java files as annotations that are then compiled into runtime checks. Both Tracematches and J-LO support parametric events, i.e., free variables can be used in the specified properties and will be bound to specific values at runtime for matching events.

The MOP framework has logic plugins, which encapsulate different logical formalisms and allow it to capture the capabilities of Tracematches and J-LO. JavaMOP is the instantiation of the MOP framework for Java programs (see Chapter 2 for an overview of the two current MOP instances).

JavaMOP allows for two different modes of matching traces, referred to as total trace matching and suffix trace matching. Total is the default mode of JavaMOP, while suffix mode is used by prefixing a JavaMOP property with the suffix modifier (see Fig. 2.4 and the accompanying text).

With total matching, for example, with the pattern $a^*b$, a sequence of events abb will trigger the validation handler of the generated MOP monitor only at the first b event and then the violation handler (if any) at the second b.

With suffix matching, however, the pattern will be matched twice, once for each b event: the first matches either the whole trace a b or the partial trace consisting of just the first b with zero occurrences of a, while the second matches the subsequent partial trace b (the second b in the trace) with zero occurrences of a; thus, the related advice will be executed twice.

With suffix matching one can count matches of a pattern open close without a need to reset the monitor after each match, as would be required with total match monitoring. On the other hand, total trace matching is more suitable for Runtime Verification of formal properties, because it is the only semantics that makes sense for some logical formalisms, such as LTL, and thus many users expect this behavior for pattern languages like regular expressions and context-free grammars, as well.

J-LO can be captured by the JavaMOP with total matching because LTL (see Section 5.4) is supported by the MOP framework. MOP supports regular expressions as part of its extended regular expression (ERE) logic plugin (see Section 5.3), and Tracematches may be captured by JavaMOP by using these ERE patterns with suffix matching.

### 1.2.2 Runtime Verification

In Runtime Verification, monitors are automatically synthesized from formal specifications, and can be deployed *offline* for debugging, or *online* for dynamically checking properties during execution. MaC [81], PathExplorer (PaX) [59], Eagle [16], and RuleR [17] are Runtime Verification frameworks for logic based monitoring, within which specific tools for Java – Java-MaC, Java PathExplorer, and Hawk [41], respectively – are implemented. All these Runtime Verification systems work in outline monitoring mode and have hardwired specification languages: MaC uses a specialized language based on interval temporal logic, JPaX supports just LTL, and Eagle adopts a fixed-point logic. Java-MaC and Java PathExplorer integrate monitors via Java bytecode instrumentation, making them difficult to port to other languages. Our MOP approach supports inline, outline, and offline monitoring; allows one to define new formalisms to extend the MOP framework; and is adaptable to new languages (e.g., Java for JavaMOP and hardware design languages for BusMOP). It is also extensible to traces predicted from viable thread interleavings, as is shown in RV-Predict.

Temporal Rover [44] is a commercial Runtime Verification tool based on future time metric temporal logic. It allows programmers to insert formal specifications in programs via annotations, from which monitors are generated. An Automatic Test Generation (ATG) component is also provided to generate test sequences from logic specifications. Temporal Rover and its successor, DB Rover, support both inline and offline monitoring. However, they also have their specification formalisms hardwired and are tightly bound to Java. MOP currently has no metric temporal logic plugin.

### 1.2.3 Design by Contract

Design by Contract (DBC) [95] is a technique allowing one to add semantic specifications to a program in the form of assertions and invariants, which are then compiled into runtime checks. It was first introduced as a built-in feature of the Eiffel language [49]. Some DBC extensions have also been proposed for a number of other languages. Jass [18] and jContractor [77] are two Java-based approaches.

Jass is a precompiler which turns the assertion comments into Java code. Besides the standard DBC features such as pre-/post- conditions and class invariants, it also provides refinement checks. The design of trace assertions in Jass is mainly influenced by CSP [64], and the syntax is more like a programming language.

jContractor is implemented as a Java library which allows programmers to associate contracts with any Java class or interface. Contract methods can be included directly within the Java class or written as a separate contract class. Before loading each class, jContractor detects the presence of contract code patterns in the Java class bytecode and performs on-the-fly bytecode instrumentation to enable checking of contracts during the program's execution. jContractor also provides a support library for writing expressions using predicate logic quantifiers and operators such as *Forall*, *Exists*, *suchThat*, and *implies*. Using jContractor, the contracts can be directly inserted into the Java bytecode even without the source code.

Java modeling language (JML) [85] is a behavioral interface specification language for Java. It provides a more comprehensive modeling language than DBC extensions. Not all features of JML can be checked at runtime; its runtime checker supports a DBC-like subset of JML. Spec# [13] is a DBC-like extension of the object-oriented language C#. It extends the type system to include non-null types and checked exceptions and also provides method contracts in the form of pre- and post-conditions as well as object invariants. Using the Spec# compiler, one can statically enforce non-null types, emit run-time checks for method contracts and invariants, and record the contracts as metadata for consumption by downstream tools.

We believe that the logics of assertions/invariants used in DBC approaches fall under the uniform format of our logic engines, so that an MOP environment following our principles would naturally support monitoring DBC specifications as a special methodological case. In addition, the MOP framework also supports outline monitoring, which we find important in assuring software reliability (e.g., monitoring for and detecting and fixing deadlocks) but which is not provided by any of the current DBC approaches that we are aware of.

### 1.2.4 Other Related Approaches

Program Query Language (PQL) allows programmers to express design rules that deal with sequences of events associated with a set of related objects [89]. Both static and dynamic tools have been implemented to find solutions to PQL queries. The static analysis conservatively looks for potential matches for queries and is useful to reduce the number of dynamic checks. The dynamic analyzer checks the runtime behavior and can perform user-defined actions when matches are found. PQL has a "hardwired" specification language based on context-free grammars

(CFG) and supports only inline monitoring. CFGs can potentially express more complex languages than regular expressions, so in principle PQL can express more complex safety policies than Tracematches. The MOP CFG plugin described in Chapter 6 allows the MOP framework to specify most of the properties that may be specified in PQL.

Program Trace Query Language (PTQL) [54] is a language based on SQL-like relational queries over program traces. The current PTQL compiler, Partiqle, instruments Java programs to execute the relational queries on the fly. PTQL events are timestamped and the timestamps can be explicitly used in queries. PTQL queries can be arbitrarily complex and, as shown in [54], PTQL's runtime overhead seems acceptable in many cases but we were unable to obtain a working package of PTQL and compare it in our experiments with JavaMOP because of license issues. PTQL properties are globally scoped and their running mode is inline. PTQL provides no support for recovery, its main use being to detect errors.

### 1.2.5 Hardware Based Monitoring

The PSL to Verilog compiler, P2V [88], is the sole attempt to perform runtime monitoring of *formal properties* in hardware, other than our BusMOP instance (see Chapters 2 and 4), of which we are aware. P2V is similar to BusMOP in that monitors are implemented in hardware rather than software, and that both approaches thus have no runtime overhead on the CPU. P2V, however, is more like the above approaches in that it is designed for monitoring actual programs rather than peripheral devices. Also it requires a dynamically extensible soft-core processor implemented on an FPGA, while our approach can potentially be applied to any COTS communication architecture. Further, P2V uses hardwired logic (PSL) while BusMOP allows different formalisms.

### 1.2.6 Predictive Analysis

There are several other approaches aimed at detecting potential concurrency errors by examining particular execution traces. Some approaches attempt to verify general purpose properties [110, 112], including temporal ones, and are inspired from debugging distributed systems based on Lamport's *happens-before* causality [84]. Other approaches work with particular properties, such as data-races and/or atomicity. [108] introduced the first lock-set based algorithm to detect data-races

dynamically, followed by many variants aiming at improving its accuracy. For example, an ownership model was used in [124] to achieve a more precise race detection at the object level. [97] combines lock-sets with happen-before. [52] provides a race detector that is both efficient and precise by switching between what they call *epochs* and vector clocks as necessary. However, their technique works only for race detection, not generic properties. Numerous other race-detection techniques have been introduced, but we feel these are only tangentially related to this work.

Previous efforts tend to focus on either soundness or coverage: those based on happens-before try to be sound, but have limited coverage over interleavings, thus missing errors; lock-set based approaches have better coverage but suffer from false alarms. Our technique aims to improve coverage without giving up soundness or genericity of properties. The only previous system to have the same features is jPredictor [34, 37], on which RV-Predict is based. As mentioned, jPredictor does not work on real sized programs due to the very limited working sets it can handle, additionally, it offers no algorithm for generic property detection.

### 1.2.7 Discussion

All this research and associated tools show that runtime monitoring is an increasingly accepted, powerful, and beneficial approach for developing reliable software and hardware. Here we summarize the systems discussed above, and show how they may be classified in terms of the five orthogonal attributes of the MOP framework: programming language, logic, scope, running mode, and handlers. The programming language determines what language the programs to be monitored must be written in. The logic specifies which formalism is used to specify the property. The scope determines where to check the property; it can be class invariant, global, interface, etc. The running mode denotes where the monitoring code runs; it can be inline (weaved into the code), online (operating at the same time as the program), outline (receiving events from the program remotely, e.g., over a socket), or offline (checking logged event traces).[2] The handlers specify what actions to perform under exceptional conditions; there can be violation and validation handlers. It is worth noting that for many logics, violation and validation are not complementary to each other, i.e., the violation of a formula does not always imply the validation of the negation of the formula.

[2]Offline implies outline, and inline implies online.

| Approach | Language | Logic | Scope | Mode | Handler |
|---|---|---|---|---|---|
| Hawk [41] | Java | Eagle | global | inline | violation |
| J-Lo [23] | Java | ParamLTL | global | inline | violation |
| Jass [18] | Java | assertions | global | inline | violation |
| JavaMaC [81] | Java | PastLTL | class | outline | violation |
| jContractor [77] | Java | contracts | global | inline | violation |
| JML [85] | Java | contracts | global | inline | violation |
| JPaX [59] | Java | LTL | class | offline | violation |
| P2V [88] | C, C++ | PSL | global | inline | validation/ violation |
| PQL [89] | Java | PQL | global | inline | validation |
| PTQL [54] | Java | SQL | global | outline | validation |
| Spec# [13] | C# | contracts | global | inline/ offline | violation |
| RuleR [17] | Java | RuleR | global | inline | violation |
| Temporal Rover [44] | C, C++, Java, Verilog, VHDL | MiTL | class | inline | violation |
| Tracematches [10] | Java | Reg. Exp. | global | inline | validation |

Figure 1.1: Runtime Monitoring Breakdown.

Most runtime monitoring approaches can be framed in terms of these attributes, while in the MOP framework they may be configured. Fig. 1.1 lists the attributes for most of the software monitoring systems discussed above. For example, JPaX can be regarded as an approach that uses linear temporal logic (LTL) to specify class-scoped properties, whose monitors work in offline mode and only detect violation.

This observation essentially motivates the design discipline of the MOP framework and specification language, namely that one should be allowed to choose the most appropriate logic and the most efficient monitoring algorithm for her/his own applications: while programming languages are designed and intended to be universal, logics and specifications tend to work best when they are domain-specific.

### 1.2.8 Examples

We next show examples for JavaMOP and BusMOP. Because RV-Predict uses JavaMOP for its handling of general specifications, the JavaMOP example can also be considered as an RV-Predict example. The difference is that, for RV-Predict, the

```
full−binding connected SafeEnum(Vector v, Enumeration e) {
 Vector instanceV;
 Enumeration instanceE;
 event createE after(Vector v) returning(Enumeration e) :
        call(* Vector.elements()) && target(v)
  {instanceE = e; instanceV = v;}
 event updateV after(Vector v) :
       (call(* Vector.add * (..)) || call(* Vector.remove(..))) && target(v)
  {instanceV = v;}
 event useE after(Enumeration e) :
        call(* Enumeration.nextElement()) && target(e)
  {instanceE = e;}
 fsm :
  start [
   updateV −> start
   createE −> enumCreated
  ]
  enumCreated [
   useE −> enumCreated
   updateV −> invalidEnum
  ]
  invalidEnum [
   updateV −> invalidEnum
  ]
 @fail {
  System.out.println("Enumeration " + __MONITOR.instanceE
           + " created from Vector " + __MONITOR.instanceV
           + " not used properly at " + __LOC);
 }
}
```

Figure 1.2: A JavaMOP Specification (SAFEENUM)

specification will be predicted within a causal cone, rather than monitored from a single, flat trace of a program.

Fig. 1.2 shows an example specification using JavaMOP; recall that this is the MOP instance for Java programs (see Chapters 2 and 3). Detailed explanation of the specification syntax can be found in Sections 2.5.1 and 2.5.2. This specification, called SafeEnum, describes the correct behavior of using Enumerations in Java. Essentially, this specification requires that an Enumeration created from a Vector not be used if the Vector has been updated since the Enumeration was created. This is important in legacy code that still uses Vectors and Enumerations because Java does not warn of this practice, it simply allows for non-deterministic results.

The specification is composed of five parts. The first line is the header of the specification, starting with two modifiers, full-binding and connected; the first states that monitor instances for this property should only raise failures when every parameter for the monitor instance has been bound (Section 3.4.1 of Chapter 3), the second states that the objects bound to the parameters must be connected by an event that actually occurs (Section 3.4.1). An ID for the specification is given after modifiers and followed by parameters of the property; in this example, two parameters are used, namely a Vector object v and an Enumeration object e.

The second part contains the declaration of two monitor variables: instanceV and instanceE. Each monitor instance for each instantiation of the specification parameters has distinct monitor instance variables. Thus, they can be used for many purposes: logging, extra states for monitoring, statistics, and so on. Here, they are used for bug reporting, to keep track of which Vector and Enumeration cause the failure.

The third part of the specification contains event declarations. Three events are defined: createE for the creation of an Enumeration, updateV for updates to a Vector, and useE for uses of an Enumeration. JavaMOP borrows (and extends; see Section 2.5.2) the syntax of AspectJ [79] for event declarations. For example, the createE event is declared to occur "after" a function call to the elements() method of class Vector. Note that the target clause is used to bind parameters in the event. Each event also sets one or both of the monitor variables, which will, again, be distinct for each binding of the parameters, using an *event action* (the Java code within the curly braces).

The fourth part of the specification is a formal description of the desired property. As discussed in Chapter 2, MOP is specification formalism independent, and one may choose different logics to specify properties. In this example, the

property description begins with fsm, meaning that a finite state machine (FSM) is used, and continues with a finite state description of the monitor. Monitors for FSM properties are initially in the first state listed in the specification, in this case start. The monitor stays in the start state until an Enumeration is created from a given Vector. Once the Enumeration has been created, it is safe to use the Enumeration until such time as the underlying Vector is modified, at which point the invalidEnum state is entered. Using an Enumeration in the invalidEnum state will result in a failure of the property.

The last part of the specification consists of handlers to execute in different states of the corresponding monitor, such as pattern match or failure. In Fig. 1.2, the handler starts with @fail, defining the action, a simple warning in this case, to execute when the trace fails to match the pattern. The handler reports which Vector and Enumeration are used incorrectly, and the line number where the failure occurs (given by the MOP-reserved variable __LOC). The __MONITOR keyword is resolved to the monitor object by JavaMOP. This is needed because there is no way from the context to tell if a given variable reference refers to a variable declared locally or a monitor instance variable.

JavaMOP specifications are compiled into AspectJ [79] aspects. Specifications as short as the one in Fig. 1.2 compile into several hundred lines of AspectJ code. The generated aspect can then be weaved into a program one wishes to monitor, using any AspectJ compiler. Once weaved, simply running the program as normal results in a monitored run of the program.

Fig. 1.3 shows an example specification using BusMOP, the MOP instance for PCI Bus monitoring (see Chapters 2 and 4). The main use for this instance is ensuring the proper use of peripherals connected to the PCI Bus. Improper use of peripherals may result from bugs in drivers or from misuse of the drivers by application programs. This specification, SafeCounterModify, states a desired property of the PCI703A digital-to-analog and analog-to-digital converter PCI board (ADC) [47]. The ADC has counters that are used to determine when input data is fully converted and ready to be placed on the PCI bus. The specification in Fig. 1.3 is concerned with the ADC's Counter 2. It requires that any modification to cntr_cntrl2, the control register on the ADC for Counter 2, happens only while the Counter 2 is not enabled (running). Counter 2 is enabled when the 0'th bit of cntr_cntrl2 is set to '1'.

As in Fig. 1.2, the first line is the header of the specification. The keyword pci specifies that this property should generate bus listening code for the PCI bus.

```
pci SafeCounterModify{
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  event countDisable : memory write address = base1 + X"220"
                       dbyte value(0) in '0'
  event cntrlMod : memory write address in base1 + X"220"
     {cntrlOld <= cntrlCurrent; cntrlCurrent <= value(15 downto 0);}
  event countEnable : memory write address = base1 + X"220"
                      dbyte value(0) in '1'
  ere : ((countEnable countDisable) | cntrlMod | countDisable)*
  @fail {
    mem_reg <= '1';
    address_reg <= base1 + X"220";
    value_reg(15 downto 0) <= cntrlOld;
    cntrlCurrent <= cntrlOld;
    enable_reg <= "0011";
  }
}
```

Figure 1.3: A BusMOP Specification (SAFECOUNTERMODIFY)

Again an ID naming the specification is provided. This time, because BusMOP does not have parameters, there is no parameter list.

The second part of the specification declares two signals, cntrlCurrent and cntrlOld, much like the monitor variables of Fig. 1.2, but BusMOP has no monitor instances, so there is only one copy of the variables. These variables are used to store the previous value of cntr_cntrl2, which is the control register for Counter 2 on the ADC board. This is necessary because PCI bus properties cannot prevent incorrect behavior, but only detect and correct it. The stored value is used to restore the value of the register when the pattern fails to match (see below).

The third part of the specification contains event declarations, much like those in Fig. 1.2, but using an instrumentation language specific to PCI Bus traffic, rather than AspectJ. Three events are defined. The keyword dbyte used in each event tells BusMOP that the quantity will be 16 bits wide (i.e., double byte). Event countDisable occurs when cntr_cntrl2, which is address X"220" in the address space of the ADC (base1 contains the address of the beginning of the ADC's address space), has its 0th bit (value(0)) set to '0', which disables Counter 2. The third event, countEnable, is analogous, but, as mentioned earlier, the bit is set to '1'. The event cntrlMod occurs when cntr_cntrl2 is modified. The keyword in is used rather than = to define the address for cntrlMod. This is because when no value for the read or write is specified, it is possible to check a whole range of addresses.

Note that this event overlaps with countDisable and countEnable. The order of the events in Fig. 1.3 is significant because simultaneous events are handled by reporting them in the declared order (see Chapter 4). Each cntrlEnable saves the previous value of the register, so that it may be restored if the property is violated. The special variable value refers to the value of the data on the bus. A pipeline is kept where the previous value is stored to cntrlOld before cntrlCurrent receives the new bus value, so that the previous value may be recovered if the pattern fails (the event action occurs before the pattern is checked).

As in Fig. 1.2, the fourth part is a formal description of the desired property, this time using an extended regular expression (ERE). This pattern specifies the desired behavior where all modifications must happen *after* disabling the counter (note again the order of event declarations, which ensures that the cntrlMod encountered from a countDisable is reported *after* the cntrlMod). The pair (countEnable countDisable) enforces that no changes can be made to cntrl_cntrl2 while Counter 2 is enabled, other than disabling it.

The last part of the specification is the handler for a pattern failure, similar to SafeEnum. An assignment of '1' to the special variable mem_reg alerts the system that a memory write is eminent. The address of the write is placed in address_reg (note that it is the control register for Counter 2). The special variable value_reg is the value to be written out by the monitor, and it is given the value of cntrlOld, which stores the previous value of cntr_cntrl2. Lastly, the enable_reg is specific to the PCI Bus interface (see Chapter 4).

BusMOP specifications are compiled into hardware description language (HDL). As in JavaMOP, the size of the generated code is far greater than that of the original specification. The HDL code is compiled into an FPGA bitstream and programmed onto an FPGA that is inserted into an empty slot on the PCI bus of the system one wishes to monitor.

The examples given in Figs. 1.2 and 1.3 may monitor completely different properties in completely different problem domains, but they follow the same pattern and philosophy. By a clear separation of monitor generation and monitor integration, MOP provides fundamental and generic support for effective and efficient application of runtime monitoring in different problem domains, and can be understood from at least three perspectives:

1. As a discipline allowing one to improve safety, reliability and dependability of a system by monitoring its requirements against its implementation at

runtime;

2. As an extension of programming languages with logics. One can add logical statements anywhere in the program, referring to past or future states of the program. These statements are like any other programming language boolean expressions, so they give the user a maximum of flexibility on how to use them: to terminate the program, guide its execution, recover from a bad state, add new functionality, etc.;

3. As a lightweight formal method. While firmly based on logical formalisms and mathematical techniques, MOP's purpose is not program verification. Instead, the idea is to avoid verifying an implementation against its specification before operation, by not letting it go wrong at runtime.

RV-Predict demonstrates that the results of instances of MOP can be used beyond simple monitoring purposes, by adapting the monitor output of JavaMOP for prediction purposes.

# Chapter 2

# Design of the MOP Framework

## 2.1 Chapter Introduction

This chapter concerns itself with the design of the MOP framework, beginning with an overview of the framework, followed by a brief description of the two current instances of the framework: JavaMOP and BusMOP. Syntax for the instances is also described. RV-Predict fits in as a client of JavaMOP, rather than as a part of this framework, and will not be discussed until Chapter 8.[1]

All monitoring systems share some features, such as program instrumentation and monitor integration, even when they aim at different domains or goals. MOP separates monitor generation and integration and provides a generic, extensible framework for runtime monitoring, allowing one to instantiate MOP with specific programming languages and specification formalisms to support different domains. In this section, we focus on the overall architecture of MOP.

### 2.1.1 Chapter Contributions

This chapter details the design of the Monitoring-Oriented Programming (MOP) framework. From a theoretical standpoint, this is the only Runtime Verification framework that is generic, not only in logical formalism, but also in terms of platform (e.g., Java applications, PCI bus monitor, System-on-Chip design).

## 2.2 Architecture

Fig. 2.1 shows the architecture of MOP. There are two kinds of high level components in MOP, namely the logic repository and language clients. The logic

[1]Work in this chapter is collaboration with Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. BusMOP is collaboration with Rodolfo Pellizzoni and Marco Caccamo. The majority of this chapter's text is modified from [93].

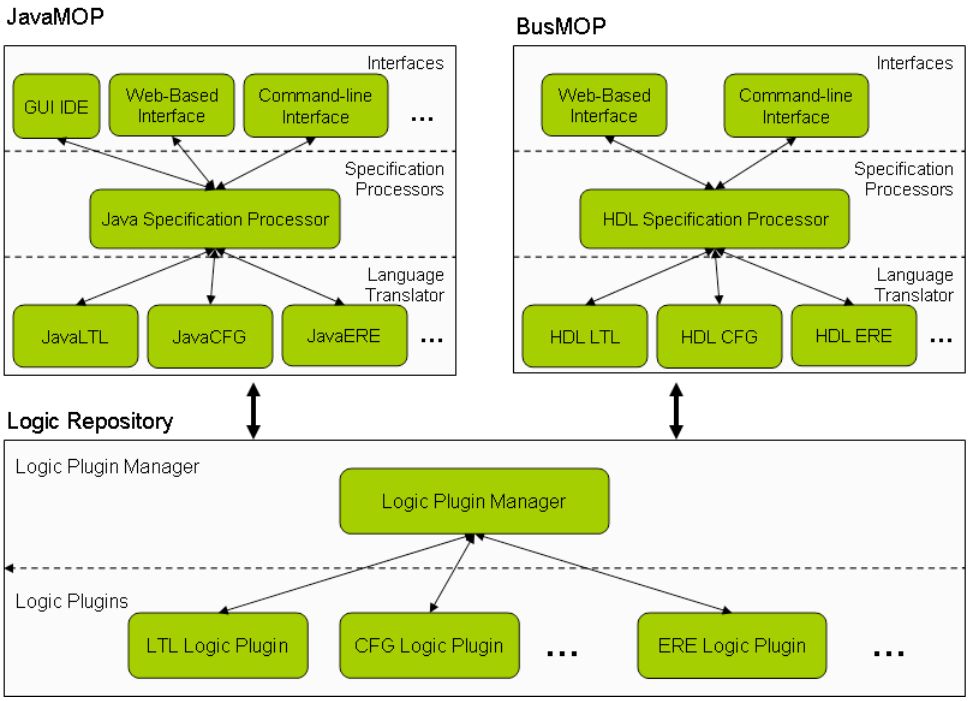



Figure 2.1: Architecture of MOP

repository, shown in the bottom of Fig. 2.1, contains various logic plugins and a logic plugin manager component. The logic plugin is the core component to generate monitoring code from formulae written in a specific logic; for example, the Linear Temporal Logic (LTL) plugin synthesizes state machines from LTL formulae. The output of logic plugins is usually pseudo-code and not bound to any specific programming language. This way, the essential monitoring generation can be shared by different instances of MOP using different programming languages. The logic plugin manager bridges the communication between the language clients and the logic plugin. More specifically, it receives the monitor generation request from the language client and distributes the request to an appropriate plugin. After the plugin synthesizes the monitor for the request, the logic plugin manager collects the result and sends it back to the language client. This way, one can easily add new logic plugins into the repository to support new specification formalisms in MOP without changing the language client.

## 2.3 Language Client

The language client hides the domain-independent logic repository and provides domain specific support for the different MOP instances. Because the domain client is the domain-specific portion of an MOP instance, we occasionally refer to the language client by the name of the MOP instance to which it belongs. Domain clients are responsible for all domain-specific aspects of monitoring, such as instrumentation, parametricity, online/inline/outline, modifiers, etc. They are usually composed of three layers: the bottom layer contains language translators that translate the abstract output of logic plugins into concrete code in a specific programming language; the middle layer is the specification processor, which extracts formulae from the given property specification and then instruments the generated monitoring code into the target program; finally, the top layer provides usage interfaces to the user.

We next explain in some detail the Java language client for the JavaMOP instance (which by abuse of terminology we will simply call JavaMOP). JavaMOP generates AspectJ [79] aspects from a specification. At the bottom layer, it has language translators for context-free grammars (CFGs), the pseudocode output generated by the past time linear temporal logic with calls and returns plugin, and finite state machine descriptions. All plugins not mentioned use finite state machine descriptions as an output language. At the mid level, as mentioned, the Java client instruments the program with the generated monitor code by creating a stand-alone aspect that can be weaved into the program using any AspectJ compiler, such as ajc [8]. At the top level there is a command line interface and a web-based interface. The two current MOP instances are discussed in Section 2.5 of this chapter, and, respectively, Chapters 3 (JavaMOP) and 4 (BusMOP), and the discussions essentially apply to the language clients associated with each instance.

## 2.4 Logic Plugins

Every logic plugin implements and encapsulates a monitor synthesis algorithm for a particular specification formalism, such as the past-time linear temporal logic (PTLTL) and the CFG plugins supported in the current MOP framework.[2] The logic plugin accepts, as input, a set of events and a formula or pattern written in the under-

[2]See Chapters 5, 6, and 7 for detailed descriptions of every plugin save the PTCaRet plugin, which is not described in this thesis. Details about PTCaRet can be found in [105] and [93].

```
//Declaration of monitor state
int state = 0;
static final int transition_createE[] = {2, 3, 3, 3};
static final int transition_updateV[] = {0, 1, 1, 3};
static final int transition_useE[] = {3, 3, 2, 3};
//Code for state update
state = transition_createE[state];
state = transition_updateV[state];
state = transition_useE[state];
//Code for category checks
Category_fail = state == 3;
```

Figure 2.2: Java code for the FSM in Fig. 1.2

lying formalism and outputs an abstract monitor. This abstract monitor is usually a piece of pseudocode, which checks a trace of events against the given formula.

We next explain in some detail one particular plugin, the plugin for FSM specifications. Fig. 2.2 shows the monitoring code generated by the MOP FSM plugin from the FSM specification in Fig. 1.2.

FSM monitors are simple, as one might expect. Static arrays keep the next state. There is one array for each event in the specification, as can be seen in Fig. 2.2. When an event arrives, the proper array is queried with the current state, and the next state is returned. After the state is updated, the category checks are preformed to see which handlers must run. Because the specification only checked @fail, we only have one check, which is for fail. As can be seen, fail is reached if the machine is in state 3. This code must be combined with generic code to handle the other properties of the specification, such as connectedness or full-binding, as well as the indexing system used for parametric trace slicing. The FSM plugin is described in Chapter 5.

## 2.5 MOP Instances

As one may expect, when putting together various languages and specification formalisms, each with its own syntax and semantics, consistency and syntactic separation may become a non-trivial problem. In this section we discuss the four dimensions that need to be instantiated in order to develop a new MOP instance (like JavaMOP or BusMOP), how they are instantiated, and where the boundary between the various components of an instance is. Since the semantics of the various pieces is typically implicit and not formally defined, in what follows we place emphasis on syntax.

Shared syntax

⟨*Specification*⟩ ::= {⟨*Instance Modifier*⟩} ⟨*Id*⟩ ⟨*Instance Parameters*⟩ "{"
{⟨*Instance Declaration*⟩}
{⟨*Event*⟩}
{⟨*Property*⟩
{⟨*Property Handler*⟩}
}
"}"

⟨*Event*⟩ ::= ["creation"]"event" ⟨*Id*⟩ ⟨*Instance Event Definition*⟩
"{" ⟨*Instance Action*⟩ "}"

⟨*Property*⟩ ::= ⟨*Logic Name*⟩ " : " ⟨*Logic Syntax*⟩

⟨*Property Handler*⟩ ::= "@" ⟨*Logic State*⟩ ⟨*Instance Handler*⟩

Instance-specific syntax

⟨*Instance Modifier*⟩ ::= ⟨*Id*⟩

⟨*Instance Parameters*⟩ ::= ⟨*JavaMOP Parameters*⟩ | ⟨*BusMOP Parameters*⟩ | ...

⟨*Instance Declaration*⟩ ::= ⟨*JavaMOP Declaration*⟩ | ⟨*BusMOP Declaration*⟩ | ...

⟨*Instance Event Definition*⟩ ::= ⟨*JavaMOP Event Definition*⟩ | ⟨*BusMOP Event Definition*⟩ | ...

⟨*Instance Action*⟩ ::= ⟨*JavaMOP Event Action*⟩ | ⟨*BusMOP Event Action*⟩ | ...

⟨*Instance Handler*⟩ ::= ⟨*JavaMOP Event Handler*⟩ | ⟨*BusMOP Event Handler*⟩ | ...

Logic-plugin-specific syntax

⟨*Logic Name*⟩ ::= ⟨*Id*⟩

⟨*Logic Syntax*⟩ ::= ⟨*FSM Syntax*⟩ | ⟨*ERE Syntax*⟩ | ⟨*LTL Syntax*⟩
| ⟨*PTLTL Syntax*⟩ | ⟨*CFG Syntax*⟩
| ⟨*PTCaRet Syntax*⟩ | ⟨*SRS Syntax*⟩ ...

⟨*Logic State*⟩ ::= ⟨*FSM State*⟩ | ⟨*ERE State*⟩ | ⟨*LTL State*⟩
| ⟨*PTLTL State*⟩ | ⟨*CFG State*⟩
| ⟨*PTCaRet State*⟩ | ⟨*SRS State*⟩ ...

Figure 2.3: MOP Syntax

### 2.5.1 MOP Syntax

Every MOP instance needs to instantiate the MOP framework in four dimensions: 1) a specification language based on the problem domain, which is mainly related to how one defines events in the domain; 2) a target language for generated monitors; 3) supported logic plugin specification formalisms; and 4) the handlers allowed in the specification. Two MOP instances have been implemented and experimented with at this point: JavaMOP and BusMOP. We expect to see more MOP instances in the future because many problem domains can benefit from monitoring.

Each instance of MOP uses an instance of the generic MOP syntax. The syntax of any instance of MOP can be generated by defining certain syntactic categories (non-terminals) of the MOP grammar, which can be seen in Fig. 2.3. All of the grammars used to define MOP syntax in this chapter use Extended Backus-Naur Form (EBNF) [48]. Non-terminals in the grammars are surrounded by "⟨" and "⟩". Braces ("{" and "}") enclose portions of the grammar that may appear zero or more times. Brackets ("[" and "]") enclose portions of the grammar that are optional (i.e., it may or may not appear). Concrete examples of the syntax defined below can be seen in Figs. 1.2 and 1.3.

#### Shared syntax

The following syntax constructs are shared by different MOP instances:

- ⟨*Specification*⟩ — ⟨*Specification*⟩ describes the generic MOP specification syntax which can be instantiated for MOP language instances and MOP logic plugins.

- ⟨*Event*⟩ — The ⟨*Event*⟩ declaration code allows for the definition of events, which may then be referred to in the property (see ⟨*Property*⟩ below). Event declarations can also have arbitrary code associated with them (⟨*Instance Action*⟩), which is run when the event is observed (⟨*Instance Event Definition*⟩), e.g. code to modify the program or the monitor state. For manual indication of events that can start a trace, the keyword creation is used at the beginning of each declaration.[3]

- ⟨*Property*⟩ — Every MOP specification may contain zero or more properties. A ⟨*Property*⟩ consists of a named formalism (⟨*Logic Name*⟩), followed by

[3]The creation keyword has no effect in BusMOP specifications.

a colon, followed by a property specification using the named formalism (see ⟨*Logic Syntax*⟩ below) and usually referring to the declared events. If the property is missing, then the MOP specification is called *raw*. Raw specifications are useful when no existing logic plugin is powerful or efficient enough to specify the desired property; in that case, one embeds the custom monitoring code manually within the ⟨*Instance Action*⟩ code.

- ⟨*Property Handler*⟩ — Handlers contain arbitrary code from the instance source language, and are invoked when a certain logic state (see ⟨*Logic State*⟩ below) or category is reached, e.g., match, fail, or a particular state in a finite state machine description.

**Instance-specific syntax**

The following constructs are based on the particular instance of MOP used for a particular specification. More information on the instances of MOP can be found in the remainder of this section, and Chapters 3 (JavaMOP) and 4 (BusMOP).

- ⟨*Instance Modifier*⟩ — ⟨*Instance Modifier*⟩s are specific to each language instance of MOP. Syntactically, they can be any valid identifier restricted by the given language. They change the behavior of the monitoring code.

- ⟨*Instance Parameters*⟩ — allow one to define the parameters of a parametric specification using the language corresponding to the MOP instance. Not all MOP instances are parametric (e.g., BusMOP), however, so this non-terminal may be empty.

- ⟨*Instance Declaration*⟩ — ⟨*Instance Declaration*⟩s are specific to each language instance of MOP. They allow for the declaration of monitor local variables.

- ⟨*Instance Event Definition*⟩ — ⟨*Instance Event Definition*⟩s are specific to each language instance of MOP. They define the conditions under which an event is triggered.

- ⟨*Instance Action*⟩ — An event can have arbitrary code associated with it, called an action. The action is run when the event is observed. An action can modify the program or the monitor state, and the syntax of the allowed statements are dependent upon the MOP instance in question. Typically the

statements used in actions have different variables and functions that may be referred to than handlers. This is why different non-terminals are used for actions and handlers.

- ⟨*Instance Handler*⟩ — ⟨*Instance Handler*⟩s are arbitrary code that is executed when a property handler is triggered.

**Logic-plugin-specific syntax**

The following constructs are based on the logic plugin(s) used in a particular specification. As mentioned, more information on logic plugins can be found in Chapters 5, 6, and 7.

- ⟨*Logic Name*⟩ — An identifier to indicate in which logic a property is defined.
- ⟨*Logic Syntax*⟩ — This refers to the syntax of the actual property definition, and is defined in the syntax section for each plugin.
- ⟨*Logic State*⟩ — ⟨*Logic State*⟩s are constants defined for each plugin, stating for which monitor states or categories (match, fail, etc.) a handler may be written.

### 2.5.2 The JavaMOP Instance

JavaMOP is an MOP development tool for Java, supporting several logical formalisms and a general specification language using them to describe Java program behaviors [32]. It compiles property specifications into optimized monitoring code. The generated code uses AspectJ [79], and is currently[4] program-independent. For example, a user can write a JavaMOP specification for a library. Then, JavaMOP generates monitoring code for this specification. This code can be applied to any program that uses the library.

In JavaMOP, an event corresponds to a pointcut, which an AspectJ compiler (such as ajc [8]) can use to weave monitoring code into the original program. Pointcuts include function call, function return, function begin, function end, field assignment, object creation, and more complex ones with pointcut operators, which

[4] We intend to incorporate program static analysis, such as [24, 27], to further reduce runtime overhead as future work.

| | | |
|---|---|---|
| ⟨*JavaMOP Modifier*⟩ | ::= | "*full−binding*" \| "*maximal−binding*" \| "*any−binding*" |
| | \| | "*connected*" \| "*unsynchronized*" |
| | \| | "*decentralized*" \| "*perthread*" \| "*suffix*" |
| ⟨*JavaMOP Parameters*⟩ | ::= | "(" [{⟨*JavaMOP Type*⟩ ⟨*Id*⟩ ","} ⟨*JavaMOP Type*⟩ ⟨*Id*⟩}] ")" |
| ⟨*JavaMOP Declaration*⟩ | ::= | syntax of declarations in Java |
| ⟨*JavaMOP Event Definition*⟩ | ::= | ⟨*AspectJ AdviceSpec*⟩ " : " ⟨*AspectJ Pointcut*⟩ ["&&" ⟨*JavaMOP Pointcut*⟩] |
| ⟨*JavaMOP Pointcut*⟩ | ::= | "*thread*" "(" ⟨*Id*⟩ ")" |
| | \| | "*condition*" "(" ⟨*BooleanExpression*⟩ ")" |
| | \| | ⟨*AspectJ Pointcut*⟩ |
| | \| | ⟨*JavaMOP Pointcut*⟩ "&&" ⟨*JavaMOP Pointcut*⟩ |
| ⟨*AspectJ Pointcut*⟩ | ::= | syntax of Pointcut in AspectJ |
| ⟨*AspectJ AdviceSpec*⟩ | ::= | syntax of AdviceSpec in AspectJ |
| ⟨*TypeList*⟩ | ::= | list of Exception types in Java |
| ⟨*Boolean Expression*⟩ | ::= | ⟨*Id*⟩ \| "!" ⟨*Boolean Expression*⟩ |
| | \| | ⟨*Boolean Expression*⟩ ⟨*Boolean Operator*⟩ ⟨*Boolean Expression*⟩ |
| | \| | "(" ⟨*Boolean Expression*⟩ ")" |
| ⟨*Boolean Operator*⟩ | ::= | " \|\| " \| "&&" \| " \| " \| "&" \| " == " \| "! = " |
| ⟨*JavaMOP TypeList*⟩ | ::= | "(" [{⟨*JavaMOP Type*⟩ ","} ⟨*JavaMOP Type*⟩] ")" |
| ⟨*JavaMOP Action*⟩ | := | Java statements, which may refer to monitor local variables |
| ⟨*JavaMOP Handler*⟩ | := | Java statements with additional keywords |
| ⟨*JavaMOP Type*⟩ | := | Any valid Java type |

Figure 2.4: JavaMOP Syntax

combine multiple simpler pointcuts. JavaMOP generates monitoring code for each pointcut—corresponding to an event in a JavaMOP specification—to maintain monitoring state, to check if the program conforms to the specification, and to trigger a handler if appropriate.

A system behavior can be described using one of several logical formalisms supported by JavaMOP, including all those described in Chapters 5, 6, and 7. A specification will be interpreted by the logic repository, a generic server used by all instances of MOP, and transformed into generic monitor code as mentioned in Section 2.2. JavaMOP translates the monitor pseudocode to AspectJ code. Any logic which can be translated to finite state machines (ERE, LTL, PTLTL) are reported to JavaMOP using the MOP finite state machine plugin syntax to reduce the number of translation algorithms necessary in JavaMOP (see Section 5.2 of Chapter 5).

A user can write a handler in Java for each monitoring state. There can be more monitoring states than simple match and fail, depending on logical formalism. A handler can be used for logging, recovering, blocking, or any other purpose. Since handlers are specified as arbitrary Java code, a user has quite a bit of latitude to

achieve his or her purposes.

**JavaMOP Syntax**

The syntax of JavaMOP is discussed below, as an instance of the generic MOP syntax defining the relevant modifiers and language-specific syntax (Java for declarations and event/handler actions, enriched with AspectJ for event definitions). The formal syntax can be seen in Fig. 2.4. Anything not explicitly described below can be considered to be identical to the generic MOP syntax. Note that some non-terminals such as ⟨*Event*⟩ refer to language instance specific non-terminals, which are defined below for JavaMOP.

- ⟨*JavaMOP Modifier*⟩ — The three binding modifiers refer to the different binding modes described in Section 3.4.1 of Chapter 3, the default is anybinding. The modifier *"unsynchronized"* tells JavaMOP that the monitor state needs not be protected against concurrent accesses; the default is synchronized. The unsynchronized monitor is faster, but may suffer from races on its state updates if the monitored program has multiple threads. The *"decentralized"* modifier refers to decentralized monitor indexing. The default indexing is centralized, meaning that the indexing trees needed to quickly access and garbage-collect monitor instances are stored in a common place; decentralized indexing means that the indexing trees are scattered all over the code as additional fields of objects of interest. Decentralized indexing typically yields lower runtime overhead, though it may not always work for all settings. Indexing modes are not described in this thesis, as they are solely the work of collaborators. Information regarding indexing can be found in [33, 93]. The *"perthread"* modifier causes JavaMOP to consider events from each thread as though from separate runs of the program, (i.e., one parametric monitor for each thread monitors only events from its own thread). The *"suffix"* modifier causes JavaMOP to consider a trace as matching if any suffix of that trace would match.

- ⟨*JavaMOP Parameters*⟩ and ⟨*JavaMOP Declaration*⟩ — These are ordinary Java parameters (as used in methods) and Java declarations. The former are the parameters of the JavaMOP specification and the latter are additional local monitor variables that one can access and modify in both event actions and property handlers. Each parameter from ⟨*JavaMOP Parameters*⟩ should be used in at least one event of the specification.

- ⟨*JavaMOP Action*⟩, ⟨*JavaMOP Handler*⟩, and ⟨*JavaMOP Event Definition*⟩ — ⟨*JavaMOP Action*⟩ are normal Java statements that may also refer to monitor local variables. ⟨*JavaMOP Handler*⟩, however, slightly extends Java with three special variables:

  - __RESET — a special expression (evaluates to void) that resets the monitor to its initial state, but does not affect any user defined varibles of the monitor;
  - __LOC — a string variable that evaluates to the line number generating the current event;
  - __MONITOR — a special variable that evaluates to the current monitor object, so that one can read/write monitor variables.

  Similarly, the advice used to define JavaMOP events slightly extends the AspectJ advice syntax. The ⟨*JavaMOP Event Definition*⟩ follows the AspectJ syntax except for its extension with ⟨*JavaMOP Pointcut*⟩, which can only be added in a top-level conjunct context. ⟨*AspectJ Pointcut*⟩ and ⟨*AspectJ AdviceSpec*⟩ are both standard AspectJ syntax [8]. The additional pointcuts have the following meaning:

  - *"thread"* — The thread pointcut captures the current thread and takes an identifier as a parameter. The identifier can be a class name or a variable name. For the former, the type of the captured thread should be a sub-class of the given class to trigger the event. For the latter, the captured thread is bound to the variable. The thread pointcut allows for the easy specification of properties which are parameterized by the current thread of execution.
  - *"condition"* — The condition pointcut takes a boolean expression as a parameter. An event containing a condition pointcut is not triggered if the boolean expression evaluates to false. This differs only from the if pointcut in standard AspectJ in that monitor instance variables may be used in the conditional expression.

### 2.5.3 The BusMOP Instance

BusMOP [100, 101] was designed to address the safety problem of third party consumer off-the-shelf (COTS) components. The complexity of safety critical

| | | |
|---|---|---|
| ⟨*BusMOP Modifier*⟩ | ::= | "*pci*" \| "*soc*" |
| ⟨*BusMOP Parameters*⟩ | ::= | $\epsilon$ (i.e., none) |
| ⟨*BusMOP Declaration*⟩ | ::= | syntax of declarations in VHDL |
| ⟨*BusMOP Event Definition*⟩ | ::= | " : " ⟨*Memory or IO*⟩ ⟨*Read or Write*⟩ "*address*" |
| | | " = " ⟨*Arithmetic Op*⟩ "*value*" ["(" ⟨*Index*⟩ ")"] ["*not*"] |
| | | "*in*" ⟨*Range*⟩ "{" [⟨*BusMOP Action*⟩] "}" |
| | \| | " : " ⟨*Memory or IO*⟩ ⟨*Read or Write*⟩ "*address*" |
| | | "*in*" ⟨*Range*⟩ |
| | \| | "*interrupt*" "{" [⟨*BusMOP Action*⟩] "}" |
| ⟨*Memory or IO*⟩ | ::= | "*memory*" \| "*io*" |
| ⟨*Read or Write*⟩ | ::= | "*read*" \| "*write*" |
| ⟨*Range*⟩ | ::= | ⟨*Arithmetic Op*⟩ ["," ⟨*Arithmetic Op*⟩] |
| ⟨*Arithmetic Op*⟩ | ::= | ⟨*Number*⟩ \| ⟨*ID*⟩ |
| | \| | ⟨*Arithmetic Op*⟩ " + " ⟨*Arithmetic Op*⟩ |
| | \| | ⟨*Arithmetic Op*⟩ "&" ⟨*Arithmetic Op*⟩ |
| | \| | ⟨*Arithmetic Op*⟩ " − " ⟨*Arithmetic Op*⟩ |
| | \| | "(" ⟨*Arithmetic Op*⟩ ")" |
| ⟨*Number*⟩ | ::= | ⟨*VHDL number or bitstring*⟩ |
| ⟨*ID*⟩ | ::= | ⟨*VHDL identifier*⟩ |
| ⟨*BusMOP Action*⟩ | := | Bus statements, which may refer to monitor local variables |
| ⟨*BusMOP Handler*⟩ | := | Bus statements, which may refer to monitor local variables, with additional output variables |

Figure 2.5: BusMOP Syntax

systems has grown to the point where the ability to use COTS in a safe manner is almost mandatory. Additionally, the vast majority of OS crashes in PCs are caused by faulty peripherals or their drivers. BusMOP answers both of these problems by allowing the specification and monitoring of properties with respect to PCI Bus traffic [100]. Its efficacy was further demonstrated in providing reliability in a System on a Chip (SoC) design platform [101].

In BusMOP, the events correspond to reads and writes of specified values to specified memory locations on a bus. PCI Bus interrupts are also allowed as events for the PCI monitoring version. The monitors, and the logic to extract events from bus traffic, are synthesized from hardware design language (HDL) code and programmed onto a field programmable gate array (FPGA), which is plugged into the PCI Bus, or they are integrated directly into the SoC.

BusMOP supports the ERE and PTLTL plugins of MOP (see Chapter 5). PTCaRet, CFG, and SRS have the problem of unbounded logic response time, which would cause the monitor to not meet timing constraints in some cases, and are thus not suitable for inclusion in BusMOP. It is also not clear exactly where the structured capabilities of PTCaRet and CFG are useful when considering flat

bus traffic traces. BusMOP was created before the current version of the logic repository. In the future it will support all finite MOP logics.

Handlers in BusMOP can be specified using arbitrary VHDL code. Several resources are provided for the user for use in handler code, such as serial output for logging, and the actual ability to write to the PCI Bus to perform recovery. PCI Bus recovery actions require bus arbitration to undo deleterious actions of faulty peripherals or their drivers (the SoC platform is able to stop bad operation before it occurs). This bus arbitration is the only possible overhead incurred by BusMOP, in cases of heavy Bus traffic. In the majority of systems, BusMOP can be used with no runtime overhead.

**BusMOP Syntax**

Below we discuss the BusMOP syntax. Anything not explicitly described below can be considered to be identical to the generic MOP syntax. Note that some non-terminals such as ⟨*Event*⟩ refer to language instance specific non-terminals, which are defined below for BusMOP. The grammar for the syntax can be seen in Fig. 2.5.

- ⟨*BusMOP Modifier*⟩ — Modifiers in BusMOP are used to distinguish the bus architecture to be monitored. Currently, the PIC bus and an SoC platform specific bus are supported.

- ⟨*BusMOP Parameters*⟩ and ⟨*BusMOP Declaration*⟩ — BusMOP is not parametric because there is no clear unit of parametrization. The nonterminal ⟨*BusMOP Declaration*⟩ refers to standard VHDL signal declarations. These are used to define additional local monitor variables that one can access and modify in both event actions and property handlers.

- ⟨*BusMOP Event Definition*⟩ — BusMOP event definitions use an original syntax to define interesting potential bus traffic. At the basic level there are three types of events: memory, IO, and interrupt. The SoC platform only has memory events. The last event is triggered when there is an interrupt on the bus. The first two are further subdivided into reads and writes. The difference between memory and IO is the address space of the read or write in question. This is important for correctly specifying the necessary bus enable signals in the generated code. Reads and writes can be concerned with the read or write of a specific location with a select range of values, or a read

or write to a range of locations where the value is of no concern. Specifying a range of read or write addresses is valuable for enforcing memory safety policies (such as, if the value 0xdeadbeef is written to address 0xffff0000 then allow no writes to some buffer until 0x000000 is written to 0xffff0000). In our SoC platform, predefined logical address names may be used in places of numerical ranges. ⟨*Arithmetic Op*⟩ allows for arithmetic operations combining variables and literal numbers. This is useful both for specifying monitor local variables and monitor input variables. Placing a numerical index on the keyword "*value*" indicates that one bit, specified by the index, should be checked rather than the whole value read or written. The monitor input variables hold the values of inputs to the monitor, and are as follows for PCI bus monitoring:

  - The value register holds the value of the read or write in question.
  - The address register holds the address of the read or write in question.
  - The baseN registers allow a user to specify a memory value relative to a given peripheral. Without this support monitoring would be very difficult due to the plug-and-play PCI bus interface that assigns memory spaces to peripherals at boot time.

    For SoC monitoring, the inputs are the same except there are no baseN registers.

- ⟨*BusMOP Action*⟩ — Actions are arbitrary VHDL statements that may refer to monitor local variables as well as the input variables described above in ⟨*BusMOP Event Definition*⟩. As mentioned in ⟨*Instance Action*⟩, these statements are executed when the event for which they are defined is observed.

- ⟨*BusMOP Handler*⟩ — Handlers are arbitrary VHDL statements that may refer to monitor local variables as well as the input variables described above in ⟨*BusMOP Event Definition*⟩. Additionally, there are variables the may be set in order to perform recovery actions. They are as follows for the PCI bus:

  - The io_reg is used to specify a read or write to I/O space. It is asserted as '1' to select the I/O space.
  - The mem_reg is used to specify a read or write to memory space by asserting it as '1'.

- The address_reg is used to specify the 32 bit address of a read or write.
- The value_reg is used to specify the value of a 32 bit read or write.
- The enable_reg is used to specify the byte enables for a read or write (specific to PCI, see Chapter 4).
- The serial_reg allows output of an ASCII value to a serial port for debugging.
- The stop_reg register that stops the peripheral in question from reading from or writing to the PCI bus when it is asserted as '1'.

The set of registers is slightly different for the SoC platform:

- The address_reg is the same as above.
- The value_reg is the same as above.
- The execute_reg is used to start a transfer.
- The reject_reg is used to reject a transfer. It is actually able to stop a faulty action before it occurs.
- The stop_reg is used to stop a process. It is similar to the stop_reg above.
- the reset_reg is used to reset a process.

As mentioned in ⟨*Instance Handler*⟩ these statements are executed when a property handler is triggered.

## 2.6 Chapter Conclusion

This chapter thoroughly explains the syntax and design of the Monitoring-Oriented Programming (MOP) framework. The information in this chapter eases the understanding of the rest of this thesis, and shows that Runtime Verification can be made efficient, expressive, and effective with a system that is generic both in terms of logical formalism and supported platforms.

# Chapter 3

# JavaMOP

## 3.1 Chapter Introduction

Each instance of MOP has issues specific to its domain. JavaMOP must deal with the complexities of parametric monitoring, in order to make itself useful in highly object-oriented systems. We first provide an introduction to parametric trace slicing (Section 3.2). We next cover improving the efficiency of parametric monitoring3.3 Lastly, we discuss different modes of parameter binding, which define which parameter instance monitors trigger handlers (Section 3.4.1).[1]

### 3.1.1 Chapter Contributions

This chapter introduces all of the logic independent features of the JavaMOP instance of the MOP framework. This chapter also details the formalization of parametric monitoring, which is both a contribution of this chapter, and a feature necessary to understand the remainder of the chapter. Two specific optimizations for formalism-independent parametric monitoring are presented. The enable set based optimization allows the algorithm to avoid creating monitor instances that will never be necessary, while the coenable optimization is able to determine which monitor instances have become unnecessary through the course of monitoring due to garbage collection, and thus mark them for removal by the system. Both of these optimizations vastly improve the efficiency of Runtime Verification. This chapter also showcases suffix matching and different parameter binding techniques, which are formalism-generic methods to improve the expressivity of parametric monitoring.

[1]Work in this chapter is collaboration with Dongyun Jin, Feng Chen, Dennis Griffith, and Grigore Roşu. It is primarily from [92] and [93], the work from the latter drawing much from [31].

## 3.2 Parametric Trace Slicing and Naive Monitoring

Parametric specifications are widely used in practice, particularly in object oriented languages, like Java, where we need to describe properties over a group of objects. For example, consider again the property in Fig. 1.2 from Chapter 1. Here the events are parametrized by the Vector $v$ and the Enumeration $e$. This is because we do not want uses of an Enumeration $e_1$ to be flagged as an error because of an intervening modification to Vector $v_2$, when it has Vector $v_1$ as its underlying Vector.

When monitoring a parametric specification, the observed execution trace is parametric, i.e., the events in the trace come with parameter information. For example, a possible parametric trace for the specification in Fig. 1.2 is:
updateV$\langle v \mapsto v_1 \rangle$ createE$\langle v \mapsto v_1, e \mapsto e_1 \rangle$ createE$\langle v \mapsto v_1, e \mapsto e_2 \rangle$ createE$\langle v \mapsto v_2, e \mapsto e_3 \rangle$ useE$\langle e \mapsto e_3 \rangle$ useE$\langle e \mapsto e_1 \rangle$ updateV$\langle v \mapsto v_1 \rangle$ useE$\langle e \mapsto e_1 \rangle$ useE$\langle e \mapsto e_2 \rangle$.

Every event in this trace is associated with a concrete parameter binding, such as $\langle v \mapsto v_1, e \mapsto e_2 \rangle$ that indicates that the parameters $v$ and $e$ in Fig. 1.2 are bound to concrete objects $v_1$ and $e_2$, respectively. Such a parametric trace represents a set of non-parametric traces each of which corresponds to a particular parameter binding. For example, the above trace contains eleven non-parametric traces for eleven parameter bindings (one for each of five singleton objects, and one for each element in the cross product of the singleton objects). The non-parametric trace for four of these bindings are summarized in the table below.

| Parameter binding | Non-parametric trace slice |
|---|---|
| $\langle v \mapsto v_1 \rangle$ | updateV updateV |
| $\langle v \mapsto v_1, e \mapsto e_1 \rangle$ | updateV createE useE updateV useE |
| $\langle v \mapsto v_1, e \mapsto e_2 \rangle$ | updateV createE updateV useE |
| $\langle v \mapsto v_2, e \mapsto e_3 \rangle$ | createE useE |

The second and third of these fail to match the pattern in Fig. 1.2, thus two failures are produced. It is highly non-trivial to monitor parametric specifications efficiently since there can be a tremendous number of parameter bindings during a single execution. For example, in a few experiments that we carried out, millions of parameter bindings were created [33]. Most other approaches for monitoring

parametric specifications handle parameters in a logic-specific way [6, 23, 89], that is, they extended the underlying specification formalisms with parameters and devised algorithms for the extended formalism. Such a solution results in very complicated monitor synthesis algorithms and makes it difficult to support new problem domains. In MOP, parameters are handled in a completely logical formalism independent manner and separated from the monitor synthesis process, vastly simplifying the implementation of new logic plugins. Surprisingly, this logic independent consideration of parameters turns out to be more efficient than those closely coupled systems (see Section 3.4.2) thanks to the clean separation of concerns. In the following sections we will explain parametric monitoring in more detail.

### 3.2.1 Events, Traces, Properties, and Parameters

First, we introduce the notions of event, trace and property, first non-parametric and then parametric. Trace slicing is then defined as a reduct operation that forgets the events that are unrelated to the given parameter instance. Most of this discussion is derived from [35].

**Definition 1** Let $\mathcal{E}$ be a set of (non-parametric) events, called **base events** or simply **events**. An **$\mathcal{E}$-trace**, or simply a (non-parametric) **trace** when $\mathcal{E}$ is understood or not important, is any finite sequence of events in $\mathcal{E}$, that is, an element in $\mathcal{E}^*$. If event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$ then we write $e \in w$.

**Example.** Consider again the SafeEnum policy from Fig. 1.2. $\mathcal{E} = \{$createE, updateV, useE$\}$ and execution traces corresponding to this are sequences of the form createE useE updateV createE useE, etc. For now we ignore the distinction between "good" and "bad" execution traces. □

**Definition 2** An **$\mathcal{E}$-property** $P$, or simply a (base or non-parametric) **property**, is a function $P : \mathcal{E}^* \rightarrow \mathcal{C}$ partitioning the set of traces into categories $\mathcal{C}$. It is common, but not enforced, that $\mathcal{C}$ includes "match", "fail", and "don't know" (or "?") categories. In general, $\mathcal{C}$, may be any set and is referred to as the set of *verdict categories* when it eases readability.

**Example.** Consider again Fig. 1.2. The FSM has no match category as we did not define it. The fail category is reached by "falling off the machine", that is, receiving

an event in a state for which there is no transition. For example, the trace createE updateV useE would result in the fail category. □

We next extend the above definitions to the parametric case, i.e., traces containing events that carry concrete data instantiating abstract parameters.
**Example.** Event useE is parametric in the Enumeration; if $e$ is the name of the generic Enumeration parameter and $e_1$ and $e_2$ are concrete Enumerations, then parametric useE events have the form useE$\langle e \mapsto e_1 \rangle$, useE$\langle e \mapsto e_2 \rangle$, etc. □

In what follows, let $[A \to B]$ be the set of total functions and $[A \rightharpoonup B]$ be the set of partial functions, both from $A$ to $B$.

**Definition 3 (Parametric events and traces).** Let $X$ be a set of **parameters** and let $V$ be a set of corresponding **parameter values**. If $\mathcal{E}$ is a set of base events like in Def. 1, then let $\mathcal{E}\langle X \rangle$ be the set of corresponding **parametric events** $e\langle\theta\rangle$, where $e$ is a base event in $\mathcal{E}$ and $\theta$ is a partial function in $[X \rightharpoonup V]$. A **parametric trace** is a trace with events in $\mathcal{E}\langle X \rangle$, that is, a word in $\mathcal{E}\langle X \rangle^*$.

To simplify writing, we occasionally assume the parameter values set $V$ implicit.
**Example.** A parametric trace for our property in Fig. 1.2 can be:

updateV$\langle v \mapsto v_1 \rangle$ createE$\langle v \mapsto v_1, e \mapsto e_1 \rangle$ createE$\langle v \mapsto v_1, e \mapsto e_2 \rangle$ createE$\langle v \mapsto v_2, e \mapsto e_3 \rangle$ useE$\langle e \mapsto e_3 \rangle$ useE$\langle e \mapsto e_1 \rangle$ updateV$\langle v \mapsto v_1 \rangle$ useE$\langle e \mapsto e_1 \rangle$ useE$\langle e \mapsto e_2 \rangle$.

We take the freedom to only list the parameter values when writing parameter instances, that is, $\langle v_1 \rangle$ instead of $\langle v \mapsto v_1 \rangle$. With this notation, the above trace is:

updateV$\langle v_1 \rangle$ createE$\langle v_1, e_1 \rangle$ createE$\langle v_1, e_2 \rangle$ createE$\langle v_2, e_3 \rangle$ useE$\langle e_3 \rangle$ useE$\langle e_1 \rangle$ updateV$\langle v_1 \rangle$ useE$\langle e_1 \rangle$ useE$\langle e_2 \rangle$.

As mentioned earlier, this trace induces *eleven trace slices*. The slice corresponding to $\langle v_1, e_1 \rangle$ is

updateV createE useE updateV useE □

**Definition 4** Partial functions $\theta$ in $[X \rightharpoonup V]$ are called **parameter instances**. $\theta, \theta' \in [A \rightharpoonup B]$ are **compatible** if for any $x \in \mathsf{Dom}(\theta) \cap \mathsf{Dom}(\theta')$, $\theta(x) = \theta'(x)$. We can **combine** compatible instances $\theta$ and $\theta'$, written $\theta \sqcup \theta'$, as follows:

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \text{when } \theta(x) \text{ is defined} \\ \theta'(x) & \text{when } \theta'(x) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$\theta \sqcup \theta'$ is also called the **least upper bound (lub)** of $\theta$ and $\theta'$. $\theta$ is **less informative** than $\theta'$, or $\theta'$ is **more informative** than $\theta$, written $\theta \sqsubseteq \theta'$, if for any $x \in X$, if $\theta(x)$ is defined then $\theta'(x)$ is also defined and $\theta(x) = \theta'(x)$.

**Definition 5 (Trace slicing)** Given parametric trace $\tau \in \mathcal{E}\langle X \rangle^*$ and $\theta$ in $[X \rightarrow V]$, let the **$\theta$-trace slice** $\tau{\restriction_\theta} \in \mathcal{E}^*$ be the non-parametric trace defined as:

- $\epsilon{\restriction_\theta} = \epsilon$, where $\epsilon$ is the empty trace/word, and
- $(\tau\, e\langle\theta'\rangle){\restriction_\theta} = \begin{cases} (\tau{\restriction_\theta})\, e & \text{when } \theta' \sqsubseteq \theta \\ \tau{\restriction_\theta} & \text{when } \theta' \not\sqsubseteq \theta \end{cases}$

The trace slice $\tau{\restriction_\theta}$ first filters out all the parametric events that are not relevant for the instance $\theta$, i.e., which contain instances of parameters that $\theta$ does not care about, and then, for the remaining events relevant to $\theta$, it forgets the parameters so that the trace can be checked against base, non-parametric properties. It is crucial to discard events for parameter instances that are not relevant to $\theta$ during the slicing, including those more informative than $\theta$, in order to achieve a "proper" slice for $\theta$: in our running example, the trace slice for $\langle v_1 \rangle$ should contain only updateV events and no createE or useE events.

**Definition 6** Let $X$ be a set of parameters together with their corresponding parameter values $V$, like in Def. 3, and let $P : \mathcal{E}^* \rightarrow \mathcal{C}$ be a non-parametric property like in Def. 2. Then we define the **parametric property** $\Lambda X.P$ as the property (over traces $\mathcal{E}\langle X \rangle^*$ and categories $[[X \rightarrow V] \rightarrow \mathcal{C}]$)

$$\Lambda X.P : \mathcal{E}\langle X \rangle^* \rightarrow [[X \rightarrow V] \rightarrow \mathcal{C}]$$

defined as $(\Lambda X.P)(\tau)(\theta) = P(\tau{\restriction_\theta})$ for any $\tau \in \mathcal{E}\langle X \rangle^*$ and any $\theta \in [X \rightarrow V]$. If $X = \{x_1, ..., x_n\}$ we may write $\Lambda x_1, ..., x_n.P$ instead of $(\Lambda\{x_1, ..., x_n\}.)P$. Also, if $P_\varphi$ is defined using a pattern or formula $\varphi$ in some particular trace specification formalism, we take the liberty to write $\Lambda X.\varphi$ instead of $\Lambda X.P_\varphi$.

A parametric property is therefore similar to a normal property, except that the domain is parametric traces, and the output, rather than being one category, is a mapping of parameter instances to categories. This allows the parametric property to associate an output category for each parameter instance from $[X \rightarrow V]$.

### 3.2.2 Monitors and Parametric Monitors

Here we define monitors $M$ and parametric monitors $\Lambda X.M$. Like for parametric properties, which are just properties over parametric traces, parametric monitors are also just monitors, but for parametric events and with instance-indexed states and output categories: a parametric monitor $\Lambda X.M$ is a monitor for the parametric property $\Lambda X.P$, with $P$ the property monitored by $M$ [35].

Monitors are defined as a variant of Moore machines:

**Definition 7** A **monitor** $M$ is a tuple $(S, \mathcal{E}, \mathcal{C}, 1, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, where $S$ is the set of states, $\mathcal{E}$ is the set of input events, $\mathcal{C}$ is the set of output categories, $1 \in S$ is the initial state, $\sigma$ is the transition function, and $\gamma$ is the output function. The transition function is extended to handle traces of events (i.e., $\sigma : S \times \mathcal{E}^* \to S$) the standard way.

The notion of a monitor above is often impractical. Actual implementations of monitors need not generate all the state space *a priori*, but rather on an "as needed" basis. Allowing monitors with infinitely many states is a necessity in our context. Even though only a finite number of states is reached during any given (finite) execution trace, there is, in general, no bound on how many may be reached. For example, monitors for context-free grammars have potentially unbounded stacks as part of their state. Also, as shown shortly, parametric monitors have domains of functions as state spaces, which are infinite as well.

**Definition 8** $M = (S, \mathcal{E}, \mathcal{C}, 1, \sigma, \gamma)$ is a **monitor for property** $P : \mathcal{E}^* \to \mathcal{C}$ if $\gamma(\sigma(1, w)) = P(w)$ for each $w \in \mathcal{E}^*$. Every monitor $M$ defines the property $\mathcal{P}_M$: $\mathcal{E}^* \to \mathcal{C}$ with $\mathcal{P}_M(w) = \gamma(\sigma(1, w))$; We write $\mathcal{P}_M$ to denote the property defined by $M$. Monitors $M$ and $M'$ are **equivalent**, written $M \equiv M'$ if $\mathcal{P}_M = \mathcal{P}_{M'}$.

We next define parametric monitors in the same style as the other parametric entities defined in this chapter: starting with a base monitor and a set of parameters, the corresponding parametric monitor can be thought of as a set of base monitors running in parallel, one for each parameter instance.

**Definition 9** Given parameters $X$ with corresponding values $V$ and monitor $M = (S, \mathcal{E}, \mathcal{C}, 1, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, the **parametric monitor** $\Lambda X.M$ is the monitor $([[X \rightharpoonup V] \to S], \mathcal{E}\langle X \rangle, [[X \rightharpoonup V] \to \mathcal{C}], \lambda\theta.1, \Lambda X.\sigma, \Lambda X.\gamma)$, with $\Lambda X.\sigma$ :

$[[X \rightharpoonup V] \to S] \times \mathcal{E}\langle X \rangle \to [[X \rightharpoonup V] \to S]$ and $\Lambda X.\gamma : [[X \rightharpoonup V] \to S] \to [[X \rightharpoonup V] \to \mathcal{C}]$ defined as

$$(\Lambda X.\sigma)(\delta, e\langle\theta'\rangle)(\theta) = \begin{cases} \sigma(\delta(\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \delta(\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{cases}$$
$$(\Lambda X.\gamma)(\delta)(\theta) = \gamma(\delta(\theta))$$

for any $\delta \in [[X \rightharpoonup V] \to S]$ and any $\theta, \theta' \in [X \rightharpoonup V]$.

Therefore, a state $\delta$ of parametric monitor $\Lambda X.M$ maintains a state $\delta(\theta)$ of $M$ for each parameter instance $\theta$, takes parametric events as input, and outputs categories indexed by parameter instances (one category of $M$ per instance). Intuitively, one can think of a parametric monitor as a *collection* of *"monitor instances"*. Each monitor instance, which is indexed by a parameter instance, keeps track of the state of one trace slice. The rule for $\Lambda X.\sigma$ can be read as stating that when an event with parameter instance $\theta'$ is evaluated, it updates the state for all monitor instances more informative than the instance for $\theta'$, and the instance for $\theta'$ itself, leaving all other monitor instances untouched. The rule for $\Lambda X.\gamma$ simply states that $\gamma$ is applied to a state, as normal, but the state is found by looking up the state of the monitor instance for $\theta$.

### 3.2.3 Naive Parametric Monitoring

Intuitively, the necessary steps for online monitoring of parametric properties are as follows:

1. Begin with a monitor instance for the empty parameter instance $\perp$ initialized to the start state of the monitor, 1.

2. As each event, $e\langle\theta\rangle$, arrives there are two possibilities:

   - There is already a monitor instance for $\theta$, in this case the instance is simply updated with $e$.
   - There is not already a monitor instance for $\theta$, in this case an instance is created for $\theta$. It is initialized to the state of the *most informative* $\theta'$ less informative than $\theta$. Such a $\theta$' is guaranteed to exist because we begin with a monitor instance for $\perp$, which is less informative than all other possible $\theta$'s. We also create monitor instances for every parameter instance that may be created by combining $\theta$ with previously

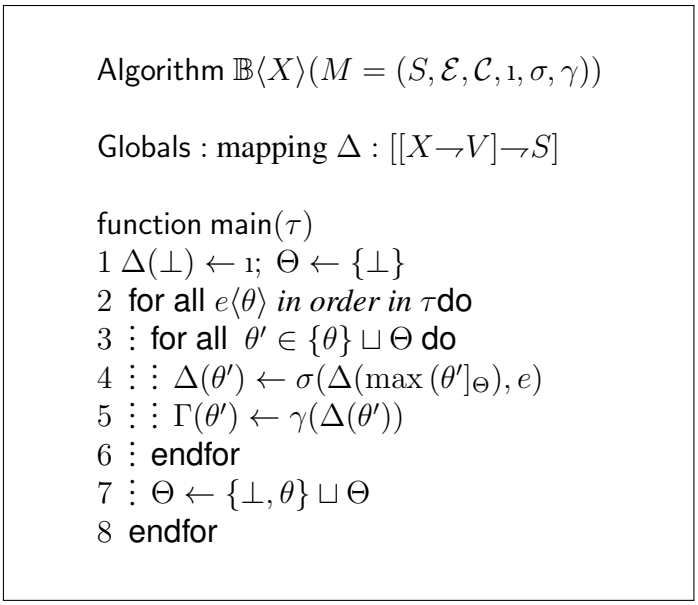



Figure 3.1: Naive Monitoring Algorithm $\mathbb{B}\langle X\rangle$

seen parameter instances. Each of these created instances is initialized similarly to the instance for $\theta$, using the most informative instance less than itself. All created monitor instances are updated with $e$ after initialization.

3. $e$ is then used to update the monitor instances for all $\theta'$ that are strictly more informative than $\theta$.

We next present a more concrete monitoring algorithm for parametric properties first introduced in [35]. It is derived from the algorithm $\mathbb{A}\langle X\rangle$, which is omitted here, that was also first presented in [35]. A first challenge here is how to represent the states of the parametric monitor. We encode the functions $[[X \rightharpoonup V] \rightharpoonup S]$ as tables with entries indexed by parameter instances in $[X \rightharpoonup V]$ and with contents states in $S$. Such tables will have finite entries since each event binds only a finite number of parameters. Fig. 3.1 shows our monitoring algorithm for parametric properties. Given parametric property $\Lambda X.P$ and $M$ a monitor for $P$, $\mathbb{B}\langle X\rangle(M)$ yields a monitor that is equivalent to $\Lambda X.M$, that is, a monitor for $\Lambda X.P$.

$\mathbb{B}\langle X\rangle$ first assigns $\imath$, the initial state, to $\Delta(\bot)$ ($\Delta$ is a mapping from parameter instance to monitor state). $\Theta$, which contains all known parameter instances is initialized to contain $\{\bot\}$, as $\bot$ is always known. For each event $e\langle\theta\rangle$ that arrives during program execution (line 2), $\mathbb{B}\langle X\rangle$ generates every compatible parameter

instance by combining $\{\theta\}$ with all the previously known parameter instances. It then updates the state of every one of these compatible parameter instances ($\theta'$) on line 4 with the state, transitioned by event $e$, of the "monitor instance" corresponding to the "largest" parameter instance less than or equal to $\theta'$. At the same time we also calculate the output corresponding to that monitor instance and store it in table $\Gamma$. Rather than storing a whole slice as in Def. 5, the knowledge of the slice is encoded in the state of the monitor instance for $\theta'$. After the algorithm completes $\Gamma$ contains the category for each possible trace slice. An actual implementation is free to report a category (e.g., match) as soon as it is discovered. In fact, in JavaMOP, it is necessary to report a category as soon as it occurs so that recovery actions can be performed, and also because the category of a trace may change several times throughout its lifetime, while $\mathbb{B}\langle X\rangle$ only gives the final result.

## 3.3 Efficient Parametric Monitoring

Algorithm $\mathbb{B}\langle X\rangle$ is correct, and easy to understand, but it is not very efficient. It creates many more monitor instances than are actually required to correctly monitor a given property. An algorithm designed for runtime monitoring should receive a trace one event at a time, rather than all at once as $\mathbb{B}\langle X\rangle$. Next we show an algorithm that receives a trace one event as a time. We also discuss optimizations to the algorithm that *vastly* improve efficiency [31].

### 3.3.1 Algorithm $\mathbb{C}\langle X\rangle$

Fig. 3.2 shows the algorithm $\mathbb{C}\langle X\rangle$[2] for online monitoring of parametric property $\Lambda X.P$, given that $M$ is a monitor for $P$. Note that we assume the notation of $\langle\rangle$ for empty maps throughout the remainder. The algorithm shows which actions to perform, e.g., creating a new monitor state and/or updating the state of related monitors, when an event is received. Algorithm $\mathbb{C}\langle X\rangle$ refines Algorithm $\mathbb{B}\langle X\rangle$ in Fig. 3.1 for efficient online monitoring. $\mathbb{C}\langle X\rangle$ essentially expands the body of the outer loop in $\mathbb{B}\langle X\rangle$ (lines 3 to 7 in Fig. 3.1). The direct use of $\mathbb{B}\langle X\rangle$ would yield prohibitive runtime overhead when monitoring large traces, because its inner loop requires searching for all parameter instances in $\Theta$ that are compatible with

[2]This algorithm was referred to as $\mathbb{C}^+\langle X\rangle$ in [31]. The distinctions between $\mathbb{C}\langle X\rangle$ and $\mathbb{C}^+\langle X\rangle$ are small, and elided for conciseness.

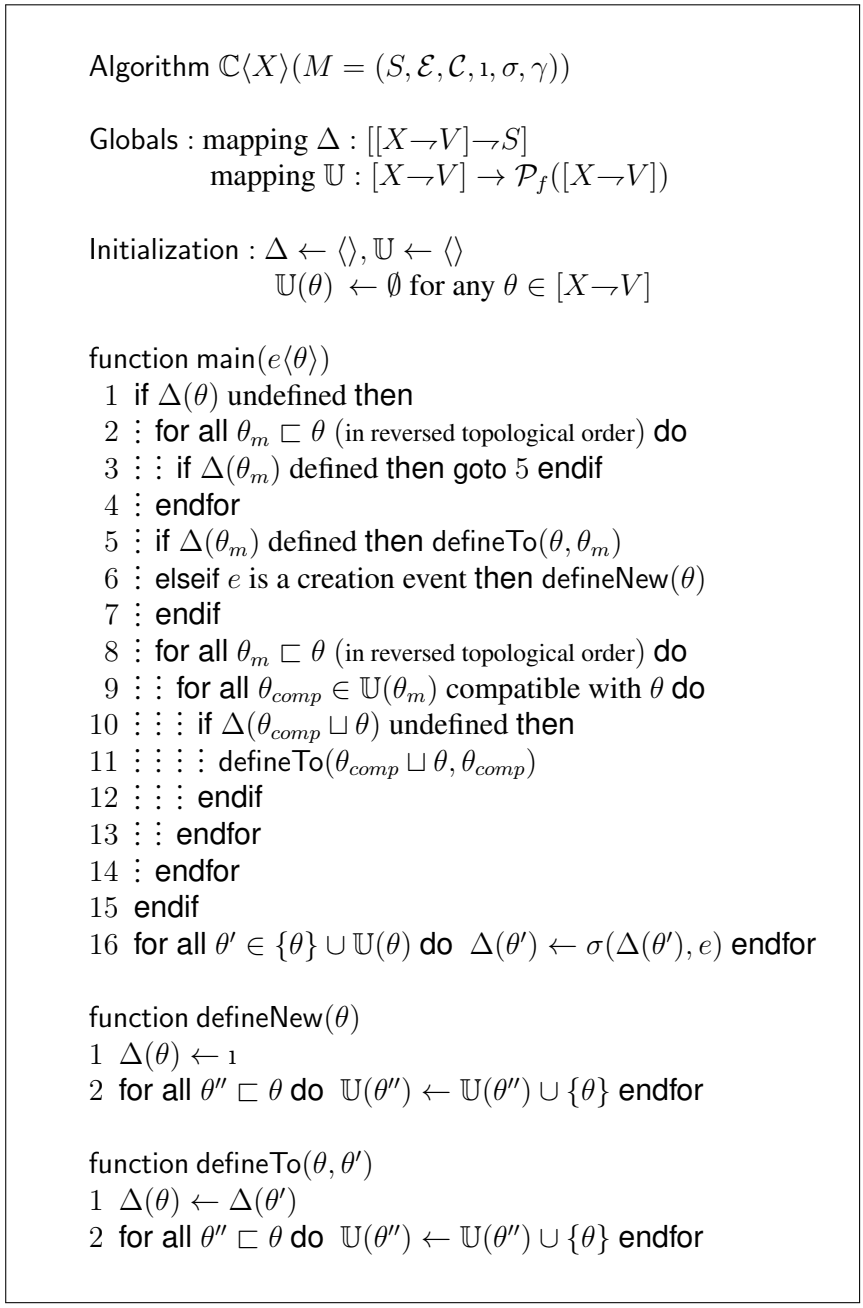



Figure 3.2: Monitoring Algorithm $\mathbb{C}\langle X\rangle$.

$$\Lambda m, c, i \;.\; \mathsf{createC}\langle m, c\rangle \; \mathsf{updateM}\langle m\rangle^* \mathsf{createI}\langle c, i\rangle$$
$$\mathsf{useI}\langle i\rangle^* \; \mathsf{updateM}\langle m\rangle^+ \; \mathsf{useI}\langle i\rangle$$

Figure 3.3: Parametric Property (UnsafeMapIterator)

$\theta$; this search can be very expensive. $\mathbb{C}\langle X\rangle$ introduces an auxiliary data structure and illustrates a mechanical way to accomplish the search, which also facilitates further optimizations.

Algorithm $\mathbb{C}\langle X\rangle$ also extends algorithm $\mathbb{B}\langle X\rangle$ to support *creation events*. Recall from Section 2.5.1 of Chapter 2 that users may specifically choose creation events using the keyword creation. Supporting *creation events* in algorithm $\mathbb{C}\langle X\rangle$ is justified and motivated by experience with implementing and evaluating $\mathbb{B}\langle X\rangle$ in [35], mainly by the following observation: one often chooses to start monitoring at the witness of a specific set of events (versus the beginning of the program).

Two mappings are used in $\mathbb{C}\langle X\rangle$: $\Delta$ and $\mathbb{U}$. $\Delta$ stores the monitor states for parameter instances, and $\mathbb{U}$ maps a parameter instance $\theta$ to *all the parameter instances* that have been defined and are properly more informative than $\theta$. In what follows, "to create a parameter instance $\theta$" and "to create a monitor state for parameter instance $\theta$" have the same meaning: to define $\Delta(\theta)$.

We next use an example about the interaction between the classes Map, Collection and Iterator in Java (Fig. 3.3), as this provides a better demonstration of the power of $\mathbb{C}\langle X\rangle$ than does the pattern in Fig. 1.2. This also provides us an opportunity to show how parametric trace slicing is truly generic with respect to the logical formalism by using extended regular expressions (ERE) in place of finite state machines (FSM). Map and Collection implement data structures for mappings and collections, respectively. Iterator is an interface used to enumerate elements in a collection-typed object. One can also enumerate elements in a Map object using Iterator. But, since a Map object contains key-value pairs, one needs to first obtain a collection object that represents the contents of the map, e.g., the set of keys or the set of values stored in the map, and then create an iterator from the obtained collection. An intricate safety property in this usage, according to the Java API specification, is that when the iterator is used to enumerate elements in the map, the contents of the map should not be changed, or unexpected behaviors may occur. The parametric LTL formula in Fig. 3.3 specifies the incorrect behavior of the system.

In Fig. 3.3, createC is an event corresponding to creating a collection from

| | $\mathsf{updateM}\langle m_1\rangle$ | $\mathsf{createC}\langle m_1, c_1\rangle$ | $\mathsf{createC}\langle m_2, c_2\rangle$ | $\mathsf{createI}\langle c_1, i_1\rangle$ |
|---|---|---|---|---|
| $\Delta$ | $\emptyset$ | $\langle m_1, c_1\rangle : \sigma(1, \mathsf{createC})$ | $\langle m_1, c_1\rangle : \sigma(1, \mathsf{createC})$<br>$\langle m_2, c_2\rangle : \sigma(1, \mathsf{createC})$ | $\langle m_1, c_1\rangle : \sigma(1, \mathsf{createC})$<br>$\langle m_2, c_2\rangle : \sigma(1, \mathsf{createC})$<br>$\langle m_1, c_1, i_1\rangle : \sigma(\sigma(1, \mathsf{createC}), \mathsf{createI})$ |
| $\mathbb{U}$ | $\emptyset$ | $\langle\rangle : \langle m_1, c_1\rangle$<br>$\langle m_1\rangle : \langle m_1, c_1\rangle$<br>$\langle c_1\rangle : \langle m_1, c_1\rangle$ | $\langle\rangle : \langle m_1, c_1\rangle, \langle m_2, c_2\rangle$<br>$\langle m_1\rangle : \langle m_1, c_1\rangle$<br>$\langle c_1\rangle : \langle m_1, c_1\rangle$<br>$\langle m_2\rangle : \langle m_2, c_2\rangle$<br>$\langle c_2\rangle : \langle m_2, c_2\rangle$ | $\langle\rangle : \langle m_1, c_1\rangle, \langle m_2, c_2\rangle, \langle m_1, c_1, i_1\rangle$<br>$\langle m_1\rangle : \langle m_1, c_1\rangle, \langle m_1, c_1, i_1\rangle$<br>$\langle c_1\rangle : \langle m_1, c_1\rangle, \langle m_1, c_1, i_1\rangle$<br>$\langle m_2\rangle : \langle m_2, c_2\rangle$<br>$\langle c_2\rangle : \langle m_2, c_2\rangle$<br>$\langle i_1\rangle : \langle m_1, c_1, i_1\rangle, \langle m_2, c_2, i_1\rangle$<br>$\langle m_1, c_1\rangle : \langle m_1, c_1, i_1\rangle$<br>$\langle m_1, i_1\rangle : \langle m_1, c_1, i_1\rangle$<br>$\langle c_1, i_1\rangle : \langle m_1, c_1, i_1\rangle$ |

Figure 3.4: Sample run of $\mathbb{C}\langle X\rangle$. The first row gives the received events; the second and the third rows give the content of $\Delta$ and $\mathbb{U}$, respectively, after every event is processed. Monitor states are represented symbolically in the table, e.g., $\sigma(1, \mathsf{createC})$ represents the state after the event createC.

a map, createI corresponds to creating an iterator from a collection, updateM corresponds to updating the map, and useI corresponds to using the iterator. The pattern says that a Collection is created from a Map, an Iterator is created from the Collection, the Map is updated at least once ($^+$ means one or more times), and then the Iterator is used after the update. The extra updateM$\langle m \rangle^*$ and useI$\langle c, i \rangle^*$ define places where these events are still valid. When an observed execution matches this pattern, the UnsafeMapIterator property is broken.

In Fig. 3.4, we show the contents of $\Delta$ and $\mathbb{U}$ after every event (given in the first row of the table) is processed. The observed trace is updateM$\langle m_1 \rangle$ createC$\langle m_1, c_1 \rangle$ createC $\langle m_2, c_2 \rangle$ createI$\langle c_1, i_1 \rangle$. We assume that createC is the only creation event. The first event, updateM $\langle m_1 \rangle$, is not a creation event and nothing is added to $\Delta$ and $\mathbb{U}$. The second event, createC$\langle m_1, c_1 \rangle$, is a creation event. So a new monitor state is defined in $\Delta$ for $\langle m_1, c_1 \rangle$, which is also added to the lists in $\mathbb{U}$ for $\perp$, $\langle m_1 \rangle$ and $\langle c_1 \rangle$. Note that $\perp$ is less informative than any other parameter instances. The third event createC$\langle m_2, c_2 \rangle$ is another creation event, incompatible with the second event. Hence, only one new monitor state is added to $\Delta$. $\mathbb{U}$ is updated similarly. The last event createI$\langle c_1, i_1 \rangle$ is not a creation event. So no monitor instance is created for $\langle c_1, i_1 \rangle$. It is compatible with the existing parameter instance $\langle m_1, c_1 \rangle$ (found from the list for $\langle c_1 \rangle$ in $\mathbb{U}$) introduced by the second event but not compatible with $\langle m_2, c_2 \rangle$ due to the conflict binding on c. Therefore, a new monitor instance is created for the combined parameter instance $\langle m_1, c_1, i_1 \rangle$ using the state for $\langle m_1, c_1 \rangle$ in $\Delta$. $\mathbb{U}$ is also updated to add the new instance into lists of parameter instances that are less informative.

### 3.3.2 Enable Set Optimization

While $\mathbb{C}\langle X \rangle$ is an improvement over $\mathbb{B}\langle X \rangle$, it is possible to improve $\mathbb{C}\langle X \rangle$ by making assumptions on the given monitor $M$. In other words, one may monitor properties written in any specification formalism, e.g., ERE, CFG, PTLTL etc., as long as one also provides a monitor generation algorithm for said formalism. However, this generality leads to extra monitoring overhead in some cases. It is possible to optimize monitor creation using the concept of *enable sets* [31]. Algorithms for computing enable sets can be found in Chapter 5 for finite logics and in Chapter 6 for context-free grammars. Enable sets computation for string-rewriting systems, unfortunately, is reducible to the Halting problem.

To motivate the optimization, let us continue the run in Fig. 3.4 to process one

| | $\mathsf{usel}\langle i_1 \rangle$ |
|---|---|
| $\Delta$ | $\langle m_1, c_1 \rangle : \sigma(1, \mathsf{createC})$<br>$\langle m_2, c_2 \rangle : \sigma(1, \mathsf{createC})$<br>$\langle m_1, c_1, i_1 \rangle : \sigma(\sigma(\sigma(1, \mathsf{createC}), \mathsf{createI}), \mathsf{usel})$<br>$\langle m_2, c_2, i_1 \rangle : \sigma(\sigma(1, \mathsf{createC}), \mathsf{usel})$ |
| $\mathbb{U}$ | $\langle\rangle : \langle m_1, c_1 \rangle, \langle m_2, c_2 \rangle, \langle m_2, c_2, i_1 \rangle, \langle m_1, c_1, i_1 \rangle$<br>$\langle m_1 \rangle : \langle m_1, c_1 \rangle, \langle m_1, c_1, i_1 \rangle$<br>$\langle c_1 \rangle : \langle m_1, c_1 \rangle, \langle m_1, c_1, i_1 \rangle$<br>$\langle m_2 \rangle : \langle m_2, c_2 \rangle, \langle m_2, c_2, i_1 \rangle$<br>$\langle c_2 \rangle : \langle m_2, c_2 \rangle, \langle m_2, c_2, i_1 \rangle$<br>$\langle i_1 \rangle : \langle m_2, c_2, i_1 \rangle, \langle m_1, c_1, i_1 \rangle$<br>$\langle m_2, c_2 \rangle : \langle m_2, c_2, i_1 \rangle$<br>$\langle m_2, i_1 \rangle : \langle m_2, c_2, i_1 \rangle$<br>$\langle c_2, i_1 \rangle : \langle m_2, c_2, i_1 \rangle$<br>$\langle m_1, c_1 \rangle : \langle m_1, c_1, i_1 \rangle$<br>$\langle m_1, i_1 \rangle : \langle m_1, c_1, i_1 \rangle$<br>$\langle c_1, i_1 \rangle : \langle m_1, c_1, i_1 \rangle$ |

Figure 3.5: Following the run of Fig. 3.4.

more event, $\mathsf{usel}\langle i_1 \rangle$. The result is shown in Fig. 3.5. $\mathsf{usel}\langle i_1 \rangle$ is not a creation event and no monitor instance is created for $\langle i_1 \rangle$. Since $\langle i_1 \rangle$ is compatible with $\langle m_2, c_2 \rangle$, a new monitor instance is defined for $\langle m_2, c_2, i_1 \rangle$. The monitor instance for $\langle m_1, c_1, i_1 \rangle$ is then updated according to usel because $\langle i_1 \rangle$ is less informative than $\langle m_1, c_1, i_1 \rangle$. $\mathbb{U}$ is also updated to add $\langle m_2, c_2, i_1 \rangle$ to the lists for all the parameter instances less informative than $\langle m_2, c_2, i_1 \rangle$. New entries are added into $\mathbb{U}$ during the update since some of the less informative parameter instances, e.g., $\langle m_2, i_1 \rangle$, have not been used before this event.

Creating the monitor instance for $\langle m_2, c_2, i_1 \rangle$ is needed for the correctness of $\mathbb{C}\langle X \rangle$, but it can be avoided when more information about the program or the specification is available. For example, according to the semantics of Iterator, no event $\mathsf{createI}\langle c_2, i_1 \rangle$ will occur in the subsequent execution since an iterator can be associated to only one collection. Hence, the monitor for $\langle m_2, c_2, i_1 \rangle$ will never reach the validation state and we do not need to create it from the beginning. However, such semantic information about the program is very difficult to infer automatically. Below, we show a simpler yet effective solution to avoid unnecessary monitor creations by analyzing the specification to monitor.

When monitoring a program against a specific property, usually only a certain

subset of property categories, ($\mathcal{C}$ in Def. 2), is checked. For example, in the UnsafeMapIterator property in Fig. 3.3, the regular expression specifies a defective interaction among related Map, Collection and Iterator objects. To find an error in the program using monitoring is thus to detect matches of the specified pattern during the execution. In other words, we are only interested in the validation category of the specified pattern. Obviously, to match the pattern, for a parameter instance of parameter set {m, c, i}, createC and createI should be observed before useI is encountered for the first time in monitoring. Otherwise, the trace slice for {m, c, i} will never match the pattern. Based on this information, we next show that creating the monitor state for $\langle m_2, c_2, i_1 \rangle$ in Fig. 3.5 is not needed. When event useI$\langle i_1 \rangle$ is encountered, if the monitor state for a parameter instance $\langle m_2, c_2 \rangle$ exists without the monitor state for $\langle m_2, c_2, i_1 \rangle$, like in Fig. 3.5, it can be inferred that in the trace slice for $\langle m_2, c_2, i_1 \rangle$, only events createC and/or updateM occur before useI because, otherwise, if createI also occurred before useI, the monitor state for $\langle m_2, c_2, i_1 \rangle$ should have been created. Therefore, we can infer, when event useI$\langle i_1 \rangle$ is observed and before the execution continues, that no match of the specified pattern can be reached by the trace slice for $\langle m_2, c_2, i_1 \rangle$, that is to say, the monitor for $\langle m_2, c_2, i_1 \rangle$ will never reach the validation state.

This observation shows that the knowledge about the specified property can be applied to avoid unnecessary creation of monitor instances. This way, the sizes of $\Delta$ and $\mathbb{U}$ can be reduced, reducing the monitoring overhead. We next formalize the information needed for the optimization and argue that it is not specific to the underlying specification formalism. How this information is used is discussed in Section 3.3.3.

**Definition 10** Given $w, w_1, w_2, w_3 \in \mathcal{E}^*$ and $e, e' \in \mathcal{E}$, for trace $w = w_1\, e'\, w_2\, e\, w_3$, *i.e.*, a trace $w$ where $e'$ occurs before an occurrence of $e$, we denote the relationship between $e$ and $e'$ with respect to $w$ as $e' \rightsquigarrow_w e$. Let the **trace enable set** of $e \in \mathcal{E}$ be the function $\mathsf{enable}_w : \mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{E})$, defined as: $\mathsf{enable}_w(e) = \{e' \mid e' \rightsquigarrow_w e\}$.

Note that if $e \notin w$ then $\mathsf{enable}_w(e) = \emptyset$. The trace enable set can be used to examine whether the execution under observation may generate a particular trace of interest, or not: if event $e$ is encountered during monitoring but some event $e' \in \mathsf{enable}_w(e)$ has not been observed, then the (incomplete) execution being monitored will *not* produce the trace $w$ when it finishes. This observation can be extended to check, before an execution finishes, whether the execution can

| Event | $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ | $\mathsf{enable}^{X}_{\mathcal{G}}$ |
|---|---|---|
| createC | $\{\emptyset\}$ | $\{\emptyset\}$ |
| createI | {{createC}, {createC, updateM}} | {{m, c}} |
| useI | {{createC, createI}, {createC, createI, updateM}} | {{m, c, i}} |
| updateM | {{createC}, {createC, createI}, {createC, createI, useI}} | {{m, c}, {m, c, i}} |

Figure 3.6: Property and Parameter Enable Sets for UnsafeMapIterator.

generate a trace belonging to some designated property categories. The designated categories are called the *goal* of the monitoring.

**Definition 11** Given $P : \mathcal{E}^* \rightarrow \mathcal{C}$ and a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, the **property enable set** is defined as a function $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}} : \mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$ with $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e) = \{\mathsf{enable}_w(e) \mid P(w) \in \mathcal{G}\}$.

Intuitively, if event $e$ is encountered during monitoring but none of event sets $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e)$ has been completely observed, the (incomplete) execution being monitored will not produce a trace $w$ s.t. $P(w) \in \mathcal{G}$. For example, given the regular expression specifying the UnsafeMapIterator property in Fig. 3.3, where $\mathcal{G} = \{\mathsf{match}\}$, the second column in Fig. 3.6 shows the property enable sets of events in UnsafeMapIterator.

The property enable set provides a sound and fast way to decide whether an incomplete trace slice has the possibility of reaching the desired categories by looking at the events that have already occurred. In the above example, if a trace slice starts with createC useI, it will never reach the match category, because $\{\mathsf{createC}\} \notin \mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(\mathsf{useI})$. In such case, no monitor state need be created even when the newly observed event may lead to new parameter instances. For example, suppose that the observed (incomplete) trace is createC useI from before. At the second event, useI, a new parameter instance can be constructed, namely, $\langle m_1, c_1, i_1 \rangle$, and a monitor state $s$ will be created for $\langle m_1, c_1, i_1 \rangle$ if algorithm $\mathbb{C}\langle X \rangle$ is applied. However, since the trace slice for $s$ is createC useI, we immediately know that $s$ cannot reach state match. So there is no need to create and maintain $s$ during monitoring if match is the goal.

A direct application of the above idea to optimize $\mathbb{C}\langle X \rangle$ requires maintaining observed events for every created monitor and comparing event sets when a new

parameter instance is found, reducing the performance. Therefore, we adapt the notion of the enable set to be based on parameter sets instead of event sets.

**Definition 12** Given a property $P : \mathcal{E}^* \to \mathcal{C}$, a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, a set of parameters $X$ and a function $\mathcal{D}_\mathcal{E} : \mathcal{E} \to \mathcal{P}_f(X)$ mapping an event to its parameters, the **property parameter enable set** of event $e \in \mathcal{E}$ is defined as a function $\mathsf{enable}^X_\mathcal{G} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(X))$ as follows: $\mathsf{enable}^X_\mathcal{G}(e) = \{\cup\{\mathcal{D}_\mathcal{E}(e') \mid e' \in \mathsf{enable}_w(e)\} \mid P(w) \in \mathcal{G}\}$.

From now on, we use "enable set" to refer to "property parameter enable set" for simplicity. For example, given the regular pattern for the UnsafeMapIterator property in Fig. 3.3 and $\mathcal{G} = \{\mathsf{match}\}$; the third column in Fig. 3.6 shows the parameter enable sets of events in UnsafeMapIterator. Then, given again the trace $\mathsf{createC}\langle m_1, c_1\rangle\ \mathsf{usel}\langle i_1\rangle$, no monitor state need be created at the second event for $\langle m_1, c_1, i_1\rangle$ since the parameter instance used to initialize the new monitor state, namely, $\langle m_1, c_1\rangle$, is not in $\mathsf{enable}^X_\mathcal{G}(\mathsf{usel})$. In other words, one may simply compare the parameter instance used to initialize the new parameter instance with the enable set of the observed event to decide whether a new monitor state is needed or not. Note that in JavaMOP, the property parameter enable sets are generated from the property enable sets provided by the formalism plugin. This allows the plugins to remain totally parameter agnostic.

### 3.3.3 *Algorithm* $\mathbb{D}\langle X\rangle$

We next integrate the concept of enable sets with algorithm $\mathbb{C}\langle X\rangle$, to improve performance and memory usage [31].

Given a set of desired verdict categories $\mathcal{G}$, we are guaranteed that we can optimize the monitoring process by omitting creating monitor states for certain parameter instances when an event is received using the enable set without missing any trace belonging to $\mathcal{G}$. However, skipping the creation of monitor states may result in false alarms, i.e., a trace that is not in $\mathcal{G}$ can be reported to belong to $\mathcal{G}$. Let us consider the following example. We monitor to find matching of a regular pattern $\mathsf{e}_1\mathsf{e}_3$. Relevant events and their parameters are $\mathsf{e}_1(p), \mathsf{e}_2(q), \mathsf{e}_3(p, q)$. The observed trace is $\mathsf{e}_1\langle p_1\rangle\mathsf{e}_2\langle q_1\rangle\mathsf{e}_3\langle p_1, q_1\rangle$. Also, suppose $\mathsf{e}_1$ is the only creation event. Obviously, the trace does not match the pattern. Fig. 3.7 shows the run using the enable set optimization (i.e., not creating monitor states for parameter instances disallowed by the enable sets). Only the content of $\Delta$ is given for simplicity. At

| | $\mathsf{e}_1\langle p_1\rangle$ | $\mathsf{e}_2\langle q_1\rangle$ | $\mathsf{e}_3\langle p_1, q_1\rangle$ |
|---|---|---|---|
| $\Delta$ | $\langle p_1\rangle : \sigma(1, \mathsf{e}_1)$ | $\langle p_1\rangle : \sigma(1, \mathsf{e}_1)$ | $\langle p_1\rangle : \sigma(1, \mathsf{e}_1)$<br>$\langle p_1, q_1\rangle : \sigma(\sigma(1, \mathsf{e}_1), \mathsf{e}_3)$ |

Figure 3.7: Unsound Usage of the Enable Set.

| | $\mathsf{e}_1\langle p_1\rangle$ | $\mathsf{e}_2\langle q_1\rangle$ | $\mathsf{e}_3\langle p_1, q_1\rangle$ |
|---|---|---|---|
| $\Delta$ | $\langle p_1\rangle : \sigma(1, \mathsf{e}_1)$ | $\langle p_1\rangle : \sigma(1, \mathsf{e}_1)$ | $\langle p_1\rangle : \sigma(1, \mathsf{e}_1)$<br>$\langle p_1, q_1\rangle : \sigma(\sigma(1, \mathsf{e}_1), \mathsf{e}_3)$ |
| $\mathcal{T}$ | $\langle p_1\rangle : 1$ | $\langle p_1\rangle : 1$ | $\langle p_1\rangle : 1$ |
| disable | $\langle p_1\rangle : 2$ | $\langle p_1\rangle : 2$<br>$\langle q_1\rangle : 3$ | $\langle p_1\rangle : 2$<br>$\langle q_1\rangle : 3$<br>$\langle p_1, q_1\rangle : 4$ |

Figure 3.8: Sound Monitoring Using Timestamps.

$\mathsf{e}_1$, a monitor state is created for $\langle p_1\rangle$ since it is the creation event. At $\mathsf{e}_2$, no action is taken since $\mathsf{enable}^X_{\mathcal{G}}(\mathsf{e}_2) = \emptyset$. At $\mathsf{e}_3$, a monitor state will be created for $\langle p_1, q_1\rangle$ using the monitor state for $\langle p \mapsto p_1\rangle$ since $\mathsf{enable}^X_{\mathcal{G}}(\mathsf{e}_3) = \{\{p\}\}$. This way, $\mathsf{e}_2$ is forgotten and a match of the pattern is reported incorrectly.

To avoid unsoundness, we introduce the notion of disable stamps of events. $\mathsf{disable} : [[X \rightharpoonup V] \rightharpoonup \mathsf{integer}]$ maps a parameter instance to an integer timestamp. $\mathsf{disable}(\theta)$ gives the time when the last event with $\theta$ was received. We maintain timestamps for monitors using a mapping $\mathcal{T} : [[X \rightharpoonup V] \rightharpoonup \mathsf{integer}]$. $\mathcal{T}$ maps a parameter instance for which a monitor state is defined to the time when the original monitor state is created from a creation event. Specifically, if a monitor state for $\theta$ is created using the initial state when a creation event is received (i.e., using the defineNew function in algorithm $\mathbb{C}\langle X\rangle$), $\mathcal{T}(\theta)$ is set to the time of creation; if a monitor state for $\theta$ is created from the monitor state for $\theta'$, $\mathcal{T}(\theta')$ is passed to $\mathcal{T}(\theta)$. Fig. 3.8 shows the evolution of disable and $\mathcal{T}$ while processing the trace in Fig. 3.7.

disable and $\mathcal{T}$ can be used together to track "skipped events": when a monitor state for $\theta$ is created using the monitor state for $\theta'$, if there exists some $\theta'' \sqsubset \theta$ s.t. $\theta'' \not\sqsubset \theta'$ and $\mathsf{disable}(\theta'') > \mathcal{T}(\theta')$ then the trace slice for $\theta$ does not belong to the

Algorithm $\mathbb{D}\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma))$

Input : mapping $\text{enable}^X_{\mathcal{G}} : [\mathcal{E} \rightarrow \mathcal{P}_f(\mathcal{P}_f(X))]$

Globals : mapping $\Delta : [[X \rightarrow V] \rightarrow S]$
mapping $\mathcal{T} : [[X \rightarrow V] \rightarrow \text{integer}]$
mapping $\mathbb{U} : [X \rightarrow V] \rightarrow \mathcal{P}_f([X \rightarrow V])$
mapping disable : $[[X \rightarrow V] \rightarrow \text{integer}]$
integer mesdames

Initialization : $\mathcal{T} \leftarrow \langle\rangle, \mathbb{U} \leftarrow \langle\rangle, \text{disable} \leftarrow \langle\rangle$
$\text{disable}(\theta) \leftarrow 0$ for any $\theta$
$\mathbb{U}(\theta) \leftarrow \emptyset$ for any $\theta, \text{mesdames} \leftarrow 0$

function main$(e\langle\theta\rangle)$
1 if $\Delta(\theta)$ undefined then
2 ⋮ createNewMonitorState$(e\langle\theta\rangle)$
3 ⋮ if $\Delta(\theta)$ undefined and $e$ is a creation event then
4 ⋮ ⋮ defineNew$(\theta)$
5 ⋮ endif
6 ⋮ disable$(\theta) \leftarrow$ mesdames; mesdames $\leftarrow$ mesdames + 1
7 endif
8 for all $\theta' \in \{\theta\} \cup \mathbb{U}(\theta)$ s.t. $\Delta(\theta')$ defined do
9 ⋮ $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$
10 endfor

function createNewMonitorStates$(e\langle\theta\rangle)$
1 for all $X_e \in \text{enable}^X_{\mathcal{G}}(e)$ (in reversed topological order) do
2 ⋮ if $\text{Dom}(\theta) \not\supseteq X_e$then
3 ⋮ ⋮ $\theta_m \leftarrow \theta'$ s.t. $\theta' \sqsubset \theta$ and $\text{Dom}(\theta') = \text{Dom}(\theta) \cap X_e$
4 ⋮ ⋮ for all $\theta'' \in \mathbb{U}(\theta_m) \cup \{\theta_m\}$ s.t. $\text{Dom}(\theta'') = X_e$ do
5 ⋮ ⋮ ⋮ if $\Delta(\theta'')$ defined and $\Delta(\theta'' \sqcup \theta)$ undefined then
6 ⋮ ⋮ ⋮ ⋮ defineTo$(\theta'' \sqcup \theta, \theta'')$
7 ⋮ ⋮ ⋮ endif
8 ⋮ ⋮ endfor
9 ⋮ endif
10 endfor

function defineNew$(\theta)$
1 for all $\theta'' \sqsubset \theta$ do
2 ⋮ if $\Delta(\theta'')$ defined then return endif
3 endfor
4 $\Delta(\theta) \leftarrow \imath$; $\mathcal{T}(\theta) \leftarrow$ mesdames; mesdames $\leftarrow$ mesdames + 1
5 for all $\theta'' \sqsubset \theta$ do $\mathbb{U}(\theta'') \leftarrow \mathbb{U}(\theta'') \cup \{\theta\}$ endfor

function defineTo$(\theta, \theta')$
1 for all $\theta'' \sqsubseteq \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ do
2 ⋮ if disable$(\theta'') > \mathcal{T}(\theta')$ or $\mathcal{T}(\theta'') < \mathcal{T}(\theta')$ then
3 ⋮ ⋮ return
4 ⋮ endif
5 endfor
6 $\Delta(\theta) \leftarrow \Delta(\theta')$; $\mathcal{T}(\theta) \leftarrow \mathcal{T}(\theta')$
7 for all $\theta'' \sqsubset \theta$ do $\mathbb{U}(\theta'') \leftarrow \mathbb{U}(\theta'') \cup \{\theta\}$ endfor

Figure 3.9: Optimized Monitoring Algorithm $\mathbb{D}\langle X\rangle$.

desired verdict categories $\mathcal{G}$. Intuitively, $\mathsf{disable}(\theta'') > \mathcal{T}(\theta')$ implies that an event $e\langle\theta''\rangle$ has been encountered after the monitor state for $\theta'$ was created. But $\theta''$ was not taken into account ($\theta'' \not\sqsubset \theta'$). The only possibility is that $e$ is omitted due to the enable set and thus the trace slice for $\theta$ does not belong to $\mathcal{G}$ according to the definition of the enable set. Therefore, in Fig. 3.8, no monitor instance is created for $\langle p_1, q_1\rangle$ at $\mathsf{e}_3$ because $\mathsf{disable}(\langle q_1\rangle) > \mathcal{T}(\langle p_1\rangle)$.

The above discussion applies when the skipped event occurs after the initial creation of the monitor state. The other case, i.e., an event is omitted before the initial monitor state is created, can also be handled using timestamps. If the skipped event is not a creation event, it does not affect the soundness of the algorithm because of the definition of creation events. In the above example, if the observed trace is $\mathsf{e}_2\langle q_1\rangle\mathsf{e}_1\langle p_1\rangle\mathsf{e}_3\langle p_1, q_1\rangle$, we will ignore $\mathsf{e}_2$ and report the matching at $\mathsf{e}_3$ since $\mathsf{e}_1$ is the only creation event. It is more sophisticated (but not much different) when the skipped event is a creation event.

Based on the above discussion, we develop a new parametric monitoring algorithm that optimizes algorithm $\mathbb{C}\langle X\rangle$ using the enable set and timestamps, as shown in Fig. 3.9. This algorithm makes use of the mappings discussed above, namely, $\mathsf{enable}^X_{\mathcal{G}}$, $\Delta$, $\mathbb{U}$, $\mathsf{disable}$, and $\mathcal{T}$, and maintains an integer variable to track the timestamp. Similar to algorithm $\mathbb{C}\langle X\rangle$, when event $e\langle\theta\rangle$ is received, algorithm $\mathbb{D}\langle X\rangle$ first checks whether $\Delta(\theta)$ is defined or not (line 1 in main). If not, monitor states may be generated for new encountered parameter instances, which is achieved by function createNewMonitorStates in algorithm $\mathbb{D}\langle X\rangle$. Unlike in algorithm $\mathbb{C}\langle X\rangle$, where all the parameter instances less informative than $\theta$ are searched to find all the compatible parameter instances using $\mathbb{U}$, createNewMon- itorStates enumerates parameter sets in $\mathsf{enable}^X_{\mathcal{G}}(e)$ and looks for parameter instances whose domains are in $\mathsf{enable}^X_{\mathcal{G}}(e)$ and which are compatible with $\theta$, also using $\mathbb{U}$. The inclusion check at line 2 in createNewMonitorStates is to omit unnecessary search since if $\mathsf{Dom}(\theta) \subseteq X_e$ then no new parameter instance will be created from $\theta$. This way, createNewMonitorStat- es creates all the parameter instances from $\theta$ whenever the enable set of $e$ is satisfied using fewer lists in $\mathbb{U}$.

If $e$ is a creation event then a monitor state for $\theta$ is initialized (lines 3–5 in main). Note that $\Delta(\theta)$ can be defined in function createNewMonitorStates if $\Delta(\theta')$ has been defined for some $\theta' \sqsubset \theta$. $\mathsf{disable}(\theta)$ is set to the current timestamp after all the creations and the timestamp is increased (line 6 in main). The rest of function main in $\mathbb{D}\langle X\rangle$ is the same as in $\mathbb{C}\langle X\rangle$: all the relevant monitor states are updated according to $e$. Function defineNew in $\mathbb{D}\langle X\rangle$ first searches for a defined

sub-instance of $\theta$. If such instance exists, $\theta$ should be defined using it; otherwise, $\Delta(\theta)$ is set to the initial state. Then $\mathcal{T}(\theta)$ is set to the current timestamp, and the timestamp is incremented. Function defineTo in $\mathbb{D}\langle X\rangle$ checks disable and $\mathcal{T}$ as discussed above to decide whether $\Delta(\theta)$ can be defined using $\Delta(\theta')$. If $\Delta(\theta)$ is defined using $\Delta(\theta')$, $\mathcal{T}(\theta)$ is set to $\mathcal{T}(\theta')$. Both functions then add $\theta$ to the sets in table $\mathbb{U}$ for the bindings less informative than $\theta$, as in $\mathbb{C}\langle X\rangle$.

### 3.3.4 Coenable Set Optimization

When monitoring parametric properties, it is easy to generate a large number of monitor instances. For example, as seen in [76], the program bloat generates 1.9 million monitor instances when monitored for the UNSAFEITER property. After some time, some of these monitor instances may become unnecessary, e.g., because they have no hope of reaching a verdict category in $\mathcal{G}$. Here we show how a dual method of the "enable set" optimization of the previous section can be derived to avoid needlessly *retaining* monitors that will never trigger. Computing the coenable sets is expected to be a quick static operation in practice, because they are a function of the specification to monitor (which is expected to be small) and not of the program (which is expected to be large). In versions of JavaMOP before the coenable optimization, presented in this section, there was a large memory leak when monitoring the UNSAFEITER property. This memory leak would happen because long living Collections would cause monitor instances for dead Iterators to be retained, as it could not remove a monitor instance unless all bound parameter objects were collected.

**Definition 13** *Given $w \in \mathcal{E}^*$ and $e, e' \in w$, we let $e \rightsquigarrow_w e'$ denote that $e'$ occurs after $e$ in $w$. Let $\mathsf{coenable}_w(e) = \{e' \mid e \rightsquigarrow_w e'\}$ be the **trace coenable set** of $e$. Given property $P\colon \mathcal{E}^* \to \mathcal{C}$ and a subset of verdict categories of interest (or goal) $\mathcal{G} \subseteq \mathcal{C}$, the **property coenable set** is defined as the map $\mathsf{coenable}_{P,\mathcal{G}}\colon \mathcal{E} \to \mathcal{P}(\mathcal{P}(\mathcal{E}))$ where $\mathsf{coenable}_{P,\mathcal{G}}(e) = \{\mathsf{coenable}_w(e) \mid w \in \mathcal{E}^* \; s.t. \; P(w) \in \mathcal{G}, e \in w, \; \mathsf{coenable}_w(e) \neq \emptyset\}$ for each $e \in \mathcal{E}$.*

Intuitively, if event $e$ is encountered during monitoring, but none of the event sets of $\mathsf{coenable}_{P,\mathcal{G}}(e)$ are possible in the future, it is impossible to reach any verdict category in $\mathcal{G}$, so a monitor for $P$ observing $e$ will never trigger. We drop all $\emptyset$s from $\mathsf{coenable}_{P,\mathcal{G}}$ because they can cause monitor instances to be retained that are unnecessary. An $\emptyset$ in $\mathsf{coenable}_{P,\mathcal{G}}(e)$ means that the trace suffix consisting of

only the event $e$ can lead to a category in $\mathcal{G}$ for some trace prefix. However, our interest is in the ability to reach $\mathcal{G}$ again in the future. If there is a trace suffix that can lead to a state in $\mathcal{G}$ from $e$, then its events will be added to $\mathsf{coenable}_{P,\mathcal{G}}(e)$. If there is no trace suffix that can lead back to a state in $\mathcal{G}$, there is no reason to maintain the monitor instance after it has executed the proper handler due to the occurrence of $e$. Coenable sets can be computed by reversing the finite state machine or context-free grammar and using the enable set calculations presented in Chapters 5 and 6, respectively. Coenable sets, like enables sets, are not computable for string rewriting systems.

**Definition 14** *Given property $P\colon \mathcal{E}^* \to \mathcal{C}$, goal $\mathcal{G} \subseteq \mathcal{C}$, set of parameters $X$ and event definition $\mathcal{D}_\mathcal{E}\colon \mathcal{E} \to \mathcal{P}(X)$ (see Definition 12), the **property parameter coenable set** is defined as the map $\mathsf{coenable}^X_{P,\mathcal{G}}\colon \mathcal{E} \to \mathcal{P}(\mathcal{P}(X))$ where $\mathsf{coenable}^X_{P,\mathcal{G}}(e) = \{\mathcal{D}_\mathcal{E}(E) \mid E \in \mathsf{coenable}_{P,\mathcal{G}}(e)\}$ for each $e \in \mathcal{E}$.*

The $\mathsf{coenable}^X_{P,\mathcal{G}}$ sets tell us which parameter objects must be alive for a verdict category in $\mathcal{G}$ to be reachable. For $P =$ UNSAFEITER, $\mathcal{G} = \{\mathsf{match}\}$, and $X = \{c, i\}$, the $\mathsf{coenable}^X_{P,\mathcal{G}}$ sets are:

$$\mathsf{coenable}^X_{P,\mathcal{G}}(\mathsf{create}) = \{\{c, i\}\}$$
$$\mathsf{coenable}^X_{P,\mathcal{G}}(\mathsf{update}) = \{\{i\}, \{c, i\}\}$$
$$\mathsf{coenable}^X_{P,\mathcal{G}}(\mathsf{next}) = \{\{c, i\}\}$$

Now with the $\mathsf{coenable}^X_{P,\mathcal{G}}$ sets we can explicitly decide when a monitor instance may be collected. For example, in UNSAFEITER we know that if, at *any time*, the Iterator bound to $i$ is garbage collected, then a match can never occur because $i$ occurs in every one of the inner sets. This makes sense because the event that causes a match in the UNSAFEITER pattern is use of the Iterator. As mentioned, this situation could produce a very large memory leak in previous versions of JavaMOP where long living Collections would cause monitor instances for dead Iterators to be retained because it could not remove a monitor instance unless all bound parameter objects were collected. We prove this concept by showing that certain parameters specified by $\mathsf{coenable}^X_{P,\mathcal{G}}(e)$ for a trace $wew'$ must be able to occur in $w'$ for a verdict category to be reached.

**Theorem 1** *Consider the same assumptions as in Definition 14, and a trace slice $wew' \in \mathcal{E}^*$. If for each $Y \in \mathsf{coenable}^X_{P,\mathcal{G}}(e)$ there exists some $y \in Y$ such that $y \notin \mathcal{D}_\mathcal{E}(w')$ then $P(wew') \notin \mathcal{G}$.*

**Proof 1** *Suppose, for the sake of contradiction, that* $P(wew') \in \mathcal{G}$ *and that each* $Y \in \mathsf{coenable}^X_{P,\mathcal{G}}(e)$ *contains a* $y$ *such that* $y \notin \mathcal{D}_\mathcal{E}(w')$*. By Definition 13, because* $P(wew') \in \mathcal{G}$ *there must be some* $E \in \mathsf{coenable}_{P,\mathcal{G}}(e)$ *that contains exactly those events in* $w'$*. Then, by Definition 14, there must be* $Y \in \mathsf{coenable}^X_{P,\mathcal{G}}(e)$ *containing exactly the parameters in* $\mathcal{D}_\mathcal{E}(w')$*. Contradiction.*

**Discussion** The $\mathsf{coenable}^X_{P,\mathcal{G}}$ sets are a conservative approximation of the situations in which a monitor instance may be collected. From Definition 5 we know that an event $e$ where $x \in \mathcal{D}_\mathcal{E}(e)$ can only occur in a trace-slice $\tau{\restriction_\theta}$ if $\theta(x)$ is still alive in the system. If $\theta(x)$ has been garbage collected, there is no way for any $e$ with $x \in \mathcal{D}_\mathcal{E}(e)$ to occur in trace slice for $\theta$. This is precisely how monitoring arrives in the situation presented in Theorem 1, where all possible suffixes $w'$ of the trace slice $wew'$ do not contain at least one parameter in each set of the $\mathsf{coenable}^X_{P,\mathcal{G}}(e)$, and it becomes impossible to reach a verdict category in $\mathcal{G}$. Clearly, if it is impossible for the $\theta$ trace slice to ever reach a verdict category in $\mathcal{G}$, there is no reason to keep the monitor instance for $\theta$.

The Tracematches system uses a more precise formulation, which is similar, but based on the *state* of the monitor. Intuitively, the Tracematches garbage collection technique can be thought of as coenables sets indexed by state rather than events, but the formulation as presented in [10] is considerably different. While theirs is more precise, our empirical results, presented in Section 3.4.2, show that the coenable set technique is able to reduce memory usage in the JavaMOP system to comparable levels with Tracematches, while the JavaMOP system has considerably lower runtime overhead. More importantly, the Tracematches garbage collection technique is limited to finite logics, such as the regular expressions of Tracematches. However, our coenable approach is extensible to any underlying monitor implementation. We have a coenables sets generation algorithm for the context-free grammar plugin (see Chapter 6). A static state-based technique, such as the one used by Tracematches, could not be used for context-free properties because the state space is unbounded.

## 3.4 Suffix Matching

According to application requirements, one may want to check a property against either *the whole execution trace* or *every suffix of a trace*. Total matching has been adopted by many Runtime Verification approaches to detect pattern failures

of properties, e.g., JPaX [59] and JavaMaC [82]. Suffix matching has been used mainly by monitoring approaches that aim to find pattern matches of properties, e.g., Tracematches [10]. PQL has a skip semantics, wherein a specification is matched against the trace, but events may be skipped. A precise explanation of PQL's semantics is available in [89]. In general, experience shows that suffix matching only makes sense for pattern languages such as context-free grammars or extended regular expressions, so we only define it for the match category of the pattern languages. To define suffix and total matching, we first give a slightly different definition of properties. Note that this definition is correct, because suffix matching is only allowed for the match category of pattern languages.

**Definition 15** *An* $\mathcal{E}$-property *P, or simply a* property*, is a pair of disjoint sets (*$P_+$*,* $P_-$*) where* $P_+ \subseteq \mathcal{E}^*$ *and* $P_- \subseteq \mathcal{E}^*$*;* $P_+$ *is the set of* pattern matching traces *and* $P_-$ *is its set of* pattern failing traces

For each particular pattern formalism, one needs to associate an appropriate property to each pattern in that formalism. For example, for a CFG $G$, we let $P_G = (P_+, P_-)$ be defined as expected: $P_+$ is $L(G)$ (the language of $G$, see Section 6.2.1 of Chapter 6) and $w \in P_-$ iff $w$ is not the prefix of any $w' \in L(G)$

**Definition 16** *The* total matching semantics *of* $P$ *is a function*

$$[\![P]\!]_{total} : \mathcal{E}^* \to \{match, fail, ?\}$$

*defined as follows for each* $w \in \mathcal{E}^*$*:*

$$[\![P]\!]_{total}(w) = \begin{cases} match & \text{if } w \in P_+ \\ fail & \text{if } w \in P_- \\ ? & \text{otherwise} \end{cases}$$

*The* suffix matching semantics *of* $P$ *is a function*

$$[\![P]\!]_{suffix} : \mathcal{E}^* \to \{match, ?\}$$

*defined as follows for each* $w \in \mathcal{E}^*$*:*

$$[\![P]\!]_{\textit{suffix}}(w) = \begin{cases} \textit{match} & \textit{if there are } w_1, w_2 \textit{ such that } w = w_1 w_2 \textit{ and} \\ & \quad [\![P]\!]_{\textit{total}}(w_2) = \textit{match} \\ ? & \textit{otherwise} \end{cases}$$

As an example of where suffix matching is useful, consider the HASNEXT property. This property specifies that the Java API method hasNext must be called before every call of next for an Iterator. If we use total matching and a match handler, we must define the pattern as (using a regular expression) "(hasNext + (hasNext next))* next next", to allow for all of the hasNext events, or correct uses of next that may occur before the two temporally adjacent calls to next. If we use suffix matching, because all suffixes of the trace are tried, the pattern "next next" is sufficient, so long as the hasNext event is still defined.

Suffix matching is implemented as a logic-independent extension of JavaMOP. This extension is based on the observation that, although total matching and suffix matching have inherently different semantics, it is not difficult to support suffix matching in a total matching setting, if one maintains *a set of monitor states* during monitoring and *creates a new monitor instance at each event* (this amounts to checking the property on each suffix incrementally). However, the situation becomes more complicated when one wants to develop a logic-independent solution, since different logical formalisms can have different state representations. For example, the monitor state can be an integer when the monitor is based on a state machine or a stack such as the CFG monitor discussed in Chapter 6. Hence, our solution is to *treat every monitor as a black-box* without assumptions on its internal state. Also, instead of maintaining a set of monitor states in the monitor, we use a wrapper monitor that keeps a set of total matching monitors as its state for suffix matching. For simplicity, from now on, when we say "monitor" without specific constraints, we mean the monitor generated for total matching. When an event is received, the wrapper monitor for suffix matching operates as follows:

1. create a new monitor and add it to the "suffix matching" monitor set;
2. invoke every monitor in the monitor set to handle the received event;
3. if a monitor enters its "pattern fail" state, remove it from the monitor set;

4. if a monitor enters its "pattern match" state, report the pattern match.

The third step is used to keep the "suffix matching" monitor set small by removing unnecessary monitors. Indeed, this implements suffix matching semantics because each total monitor is monitoring a suffix of the current trace, and "pattern match" is only reported if one of the suffixes is valid.

Using our current implementation of suffix matching in JavaMOP, one may further improve the monitoring efficiency if the monitor provides an equals method that compares two monitors with regard to their internal states, and a hashCode method used to reduce the amount of calls to equals. This interface is used to populate a Java HashSet: the combination of the definition of hashCode and equals ensures the monitors in the HashSet are declared duplicates, and removed, based on monitor state rather than memory location. This interface can be easily generated by each JavaMOP logic plugin because it has full knowledge of the monitor semantics. It is important to note that our approach does *not depend on the underlying specification formalism*.

### 3.4.1 Binding Modes and Connectedness

There are three parameter binding modes available in JavaMOP, as well as the concept of connectedness, which may be used in conjunction with any binding mode. These modes and connectedness determine which monitor instances are allowed to report verdict categories (e.g., match or fail). This allows a user to essentially apply a filter on the number of results they want from their monitors. For each binding mode we will consider the pattern:

$$\Lambda a, b \,.\, \mathsf{e}_1\langle\rangle(\mathsf{e}_2\langle a, b\rangle \mid \mathsf{e}_3\langle b\rangle)^*$$

And the following trace:

$$\mathsf{e}_1\langle\rangle \;\; \mathsf{e}_2\langle a_1, b_1\rangle \;\; \mathsf{e}_3\langle b_1\rangle$$

The default binding mode is the any-binding mode. In this mode any instance monitor is allowed to report categories. When the above trace is monitored using any-binding four matches are reported, one on each of the first two events as they arrive and two for the last event, $\mathsf{e}_3$. Two matches are reported when $\mathsf{e}_3$ arrives because one is reported from the monitor instance for $\langle a_1, b_1\rangle$ and one is reported from the monitor instance for $\langle b_1\rangle$, and categories can be reported from any monitor

| Instance | Trace |
|---|---|
| $\langle\rangle$ | $\mathsf{e}_1$ |
| $\langle b_1\rangle$ | $\mathsf{e}_1$ $\mathsf{e}_3$ |
| $\langle a_1, b_1\rangle$ | $\mathsf{e}_1$ $\mathsf{e}_2$ $\mathsf{e}_3$ |

Figure 3.10: Trace Slices for Binding Mode Example Trace

instance. The trace slices for each monitor instance after all three events can be seen in Fig. 3.10. Note that each one matches the pattern, and that the one for $\langle a_1, b_1\rangle$ has *two* prefixes that match the pattern ($\mathsf{e}_1$ $\mathsf{e}_2$ and $\mathsf{e}_1$ $\mathsf{e}_2$ $\mathsf{e}_3$), resulting in four total matches, as expected.

The other extreme, full-binding is allowing only those instance monitors that correspond to fully instantiated parameter instances to report categories. This is similar to the semantics of Tracematches [6, 10, 26]. Looking at our example trace, two matches will be reported because two prefixes of the trace $\mathsf{e}_1$ $\mathsf{e}_2$ $\mathsf{e}_3$, which comes from the only instance we consider ($\langle a_1, b_1\rangle$), match (see Fig. 3.10). To implement this mode, during monitor generation we count the number of parameters in the parameter list of the monitor. Whenever we check a monitor instance for a category we compare the number of bound parameters of the monitor instance to the number of parameters of the specification. If the numbers of parameters do not match, the monitor instance's output is ignored.

In between these two extremes is maximal-binding. The "less informative or as informative as" relation "$\sqsubseteq$" in Def. 4 induces a partial order over parameter instances. In this mode we only report verdict categories from those instances that are *currently* maximal in that partial order. Considering again our example, this is a bit more complex. When event $\mathsf{e}_1$ arrives, the instance $\langle\rangle$ *is maximal*, so a match is reported. When $\mathsf{e}_2$ arrives the new maximal instance is $\langle a_1, b_1\rangle$, and a match is reported. When $\mathsf{e}_3$ arrives, $\langle a_1, b_1\rangle$ is still larger than $\langle b_1\rangle$ so only the instance $\langle a_1, b_1\rangle$ is allowed to report a match, thus only three matches are reported, unlike the four from any-binding. To implement this binding mode, all monitor instances contain flags; when defineTo in Algorithm $\mathbb{D}\langle X\rangle$ (Fig. 3.9) defines a new monitor from a less informative monitor the flag in the less informative monitor is set to false. Additionally, when a new instance is created we must check if there is a monitor instance for a more informative parameter instance already in existence, as happens in our example, and set the flag to false if there is. Results from the monitor instance are only reported when the flag is true.

| Instance | Trace |
|---|---|
| $\langle v_1 \rangle$ | updateV |
| $\langle e_1 \rangle$ | useE |
| $\langle v_2 \rangle$ | updateV |
| $\langle v_1, e_1 \rangle$ | updateV createE useE |
| $\langle v_2, e_1 \rangle$ | updateV useE |

Figure 3.11: Trace Slices for Connectedness Example Trace

Connectedness, which may be used to augment any binding mode, filters out all those monitor instances for which the parameters are not connected to each other by some event. For example, if events $\mathsf{e}_1\langle p_1 \rangle$ and $\mathsf{e}_2\langle q_1 \rangle$ are the only events that have been seen that are sent to the $\langle p_1, q_1 \rangle$ instance, no categories will be reported from that instance until some event such as $\mathsf{e}_3\langle p_1, q_1 \rangle$ occurs. For connectedness we will consider again the SafeEnum property of Fig. 1.2, and we will consider the trace:

$$\mathsf{updateV}\langle v_1 \rangle\ \mathsf{createE}\langle v_1, e_1 \rangle\ \mathsf{updateV}\langle v_2 \rangle\ \mathsf{useE}\langle e_1 \rangle$$

Nothing violating the SafeEnum condition of not using an Enumeration created from a Vector that has been modified occurs in this pattern, however, because of the generic parametric trace slicing algorithm, some monitor instances will be generated that will flag failures, as can be seen in Fig. 3.11. With connectedness the results of extraneous monitor instances such as $\langle v_2, e_1 \rangle$, which would signal an undesired fail verdict, can be filtered out. Note that the instance for $\langle v_2, e_1 \rangle$ must be created by the generic parametric monitoring algorithm, because it has no way to know that $\langle v_2, e_1 \rangle$ cannot be connected at some time in the future, as it has no semantic knowledge of the createE event. Connectedness can be added to a specification via the connected keyword shown in Fig. 2.4, and is implemented using a union-find data structure in which each set represents parameter objects which have been connected by a given event. When an event arrives, all of its associated parameter objects are unioned in the union-find data structure. When a monitor instance attempts to report a verdict category, the union-find data structure is queried to ensure that all parameter objects of the instance are in the same set of the union-find.

### 3.4.2 JavaMOP Evaluation

In this section we evaluate JavaMOP with the enable (Section 3.3.2), and coenable (Section 3.3.4) set optimizations. Additionally, several engineering improvements such as indexing caching and different indexing techniques which can be found in [76] and [33] were used. These are not mentioned in this thesis because they are the sole work of collaborators. This version of JavaMOP is compared to the previous version, which had the distributed indexing scheme, but no enable, coenable, or index caching techniques and Tracematches, the two more performant runtime monitoring systems prior to the work of this chapter.

### 3.4.3 Experimental Settings

For our experiments, we used a Pentium 4 2.66GHz / 2GB RAM / Ubuntu 9.10 machine and version 9.12 of the DaCapo (DaCapo 9.12) benchmark suite [22]. We also present results from the previous version, 2006-10 MR2, of DaCapo, but only for the benchmarks that are not included in the new version of DaCapo: antlr, bloat, chart, hsqldb, and jython. Among deprecated benchmarks that DaCapo 9.12 does not provide anymore, we particularly favor the bloat benchmark from DaCapo 2006-10 because it generates large overheads when monitoring Iterator-based properties. The bloat benchmark with the UNSAFEITER specification causes 11258% runtime overhead (i.e., 113 times slower) and uses 7.8MB of heap memory in Tracematches, and causes 769% runtime overhead and uses 175MB in the previous version of JavaMOP, while the original program uses only 4.9MB. Also, although DaCapo 9.12 provides jython, Tracematches cannot instrument jython due to an error, while all versions of JavaMOP are able to instrument it. Thus, we present the result of jython from DaCapo 2006-10. We use the default data input for DaCapo and the -converge option to obtain the numbers after convergence within $\pm 3\%$. We also tested other benchmarks including Java Grande [115] and SPECjvm 2008 [1], and saw little to no overhead even with our Iterator-based properties, so we omit the results. Instrumentation causes different garbage collection behavior in the monitored program, sometimes causing the it to slightly outperform the original program; this, as well as the fact that convergence is only within 3%, accounts for the negative overheads seen in both runtime and memory.

All experiments were performed with Sun JVM 1.6.0. The AspectJ compiler (ajc) version 1.6.4 was used for weaving the JavaMOP generated aspects. Another AspectJ compiler, abc [9] 1.3.0, was used for weaving Tracematches properties

because Tracematches is part of abc, rather than generating stand-alone aspects as does JavaMOP. For the previous version of JavaMOP, we used JavaMOP 2.1.2, which can be found at [74], but with the -noopt1 option to turn off the enable set optimization. For the new version of JavaMOP, we used the release version, 2.3.2, which can be found at the same location. For Tracematches, we used release version 1.3.0, from [121], which is included in the abc compiler as an extension. To discover why some examples do not terminate when using Tracematches, we also used the abc compiler for weaving aspects generated from JavaMOP properties. Note that JavaMOP is AspectJ compiler independent. JavaMOP shows similar overheads and terminates on all examples when using the abc compiler for weaving as when ajc is used. Because the overheads are similar, we do not present the results of using abc to weave JavaMOP generated aspects in this thesis. However, using abc to weave JavaMOP properties confirms that the high overhead and non-termination come from Tracematches itself, not from the abc compiler.

The following properties are used in our experiments. They were borrowed from [24, 26, 31, 92].

- HASNEXT: Do not use the next element in an Iterator without checking for the existence of it;
- UNSAFEITER: Do not update a Collection when using the Iterator interface to iterate its elements;
- UNSAFEMAPITER: Do not update a Map when using the Iterator interface to iterate its values or its keys;
- UNSAFESYNCCOLL: If a Collection is synchronized, then its iterator also should be accessed synchronously;
- UNSAFESYNCMAP: If a Collection is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner.

All of them are tested with Tracematches, and with the previous and new versions of JavaMOP for comparison. We have tested several non-Iterator based properties: HASHSET, SAFEENUM, SAFEFILE, and SAFEFILEWRITER [24, 26, 31, 92]. None of these properties produce overheads above 5% in any of the DaCapo benchmarks, thus their results are not presented here.

### 3.4.4 Results and Discussions

Tables 3.1, 3.2, 3.3, 3.4, 3.5, and 3.6 summarize the results of the evaluation. Note that the structure of DaCapo 9.12 allows us to instrument all of the benchmarks plus all supplementary libraries that the benchmarks use, which was not possible for DaCapo 2006-10. Therefore, fop and pmd show higher overheads than the benchmarks using DaCapo 2006-10 from [31]. While other benchmarks show overheads less than 80% in the previous version of JavaMOP, bloat, avrora, batik, and pmd show prohibitive overhead in both runtime and memory performance. This is because they generate many Iterators and all properties in this evaluation are intended to monitor Iterators. For example, bloat creates 1,625,770 Collections and 941,466 Iterators in total while 19,605 Iterators coexist at the same time at peak, in an execution. avrora and pmd also create many Collections and Iterators. Also, they call hasNext() 78,451,585 times, 1,158,152 times and 4,670,555 times and next() 77,666,243 times, 352,697 times and 3,607,164 times, respectively. Therefore, in this section, we mainly discuss those examples that have shown the most overhead for the previous version of JavaMOP, although the new version of JavaMOP shows improvements for other examples as well.

Tables 3.1, 3.2, 3.3 show the percent runtime overhead of Tracematches and the previous and new versions of JavaMOP. The previous version of JavaMOP shows, on average, 54% runtime overhead, but the optimized JavaMOP shows only 21% runtime overhead (16% except the cases where the previous version crashed due to running out of memory). This is less than half of the average runtime overhead that the previous version of JavaMOP showed. Compared to Tracematches, the optimized JavaMOP shows orders of magnitude less runtime overhead; Tracematches shows, on average, 309% runtime overhead. Even if we ignore the fact that Tracematches and the previous version of JavaMOP crashed on several cases, it clearly shows the improvements in runtime overhead when our optimization techniques are used. In the worst case benchmark program, bloat, the optimized JavaMOP had runtime overhead below 260%, while the previous JavaMOP showed more than 440% runtime overhead and Tracematches showed more than 1300%. Both Tracematches and the previous version of JavaMOP crashed when monitoring UNSAFEMAPITER. With avrora, on average, the new version of JavaMOP shows 62% runtime overhead, while the previous version of JavaMOP showed 140% runtime overhead and Tracematches showed 203%. Both of them hanged when monitoring UNSAFEMAPITER. With pmd, on average, the

| | | HASNEXT | | | UNSAFEITER | | |
|---|---|---|---|---|---|---|---|
| | ORIG (sec) | TM | Old | New | TM | Old | New |
| antlr | 3.6 | -1 | 4 | -2 | 0 | 0 | -2 |
| bloat | 14.4 | 2119 | 448 | 116 | 11258 | 769 | 251 |
| chart | 12.2 | 0 | 0 | -2 | 11 | 5 | -1 |
| hsqldb | 8.4 | 15 | 0 | -3 | 17 | -1 | -3 |
| jython | 9.0 | 13 | 0 | 0 | 11 | -4 | 1 |
| avrora | 13.9 | 45 | 48 | 55 | 637 | 298 | 118 |
| batik | 3.5 | 3 | 4 | 3 | 355 | 11 | 8 |
| eclipse | 79.5 | -2 | 1 | -1 | 0 | 2 | -1 |
| fop | 2.0 | 200 | 57 | 48 | 350 | 23 | 13 |
| h2 | 18.7 | 89 | 17 | 13 | 128 | 7 | 4 |
| luindex | 2.9 | 0 | 1 | 1 | 0 | 1 | 1 |
| lusearch | 25.3 | -1 | 7 | 0 | 1 | 0 | 2 |
| pmd | 8.4 | 176 | 89 | 59 | 1423 | 162 | 123 |
| sunflow | 32.5 | 47 | 5 | 3 | 7 | 0 | 0 |
| tomcat | 14.1 | 8 | -1 | 1 | 37 | -1 | 1 |
| tradebeans | 45.7 | 0 | 1 | 1 | 1 | 0 | 2 |
| tradesoap | 95.0 | 1 | 0 | 0 | 2 | -2 | 1 |
| xalan | 20.9 | 4 | -2 | 2 | 27 | 2 | 2 |

Table 3.1: Average *Percent* Runtime Overhead for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against HASNEXT and UNSAFEITER (convergence within 3%, OOM = Out of Memory)

| | | UNSAFEMAPITER | | | UNSAFESYNCCOLL | | |
|---|---|---|---|---|---|---|---|
| | ORIG (sec) | TM | Old | New | TM | Old | New |
| antlr | 3.6 | -2 | 5 | 1 | -1 | 2 | -1 |
| bloat | 14.4 | OOM | OOM | 178 | 1359 | 735 | 212 |
| chart | 12.2 | -1 | 4 | -2 | -2 | 1 | -1 |
| hsqldb | 8.4 | 29 | 0 | -3 | 9 | 0 | -2 |
| jython | 9.0 | 150 | 11 | 3 | 11 | -4 | 1 |
| avrora | 13.9 | OOM | OOM | 42 | 75 | 140 | 80 |
| batik | 3.5 | OOM | 65 | 5 | 208 | 444 | 9 |
| eclipse | 79.5 | 5 | -1 | 0 | -4 | -1 | 1 |
| fop | 2.0 | OOM | OOM | 14 | OOM | OOM | 25 |
| h2 | 18.7 | 1350 | OOM | 6 | 868 | 69 | 4 |
| luindex | 2.9 | 1 | 0 | 1 | 1 | 1 | 1 |
| lusearch | 25.3 | 2 | 2 | 0 | 4 | 0 | 1 |
| pmd | 8.4 | OOM | OOM | 188 | 1818 | OOM | 76 |
| sunflow | 32.5 | 9 | 6 | 1 | 13 | 6 | 5 |
| tomcat | 14.1 | 3 | -1 | 1 | 2 | -1 | 1 |
| tradebeans | 45.7 | 5 | -1 | -1 | -1 | -1 | 2 |
| tradesoap | 95.0 | 2 | 0 | 1 | 0 | 0 | 1 |
| xalan | 20.9 | 10 | 1 | 2 | 3 | 1 | 3 |

Table 3.2: Average *Percent* Runtime Overhead for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against UNSAFEMAPITER and UNSAFESYNCCOLL (convergence within 3%, OOM = Out of Memory)

| | | UNSAFESYNCMAP | | |
|---|---|---|---|---|
| | ORIG (sec) | TM | Old | New |
| antlr | 3.6 | 0 | 2 | 0 |
| bloat | 14.4 | 1942 | 858 | 130 |
| chart | 12.2 | -2 | 3 | -2 |
| hsqldb | 8.4 | 7 | -1 | -3 |
| jython | 9.0 | 10 | -4 | 0 |
| avrora | 13.9 | 54 | 73 | 16 |
| batik | 3.5 | 5 | 7 | 0 |
| eclipse | 79.5 | OOM | 2 | -1 |
| fop | 2.0 | OOM | OOM | 19 |
| h2 | 18.7 | 83 | 25 | 5 |
| luindex | 2.9 | 2 | 2 | 0 |
| lusearch | 25.3 | 3 | 1 | 1 |
| pmd | 8.4 | OOM | OOM | 26 |
| sunflow | 32.5 | 17 | 8 | 6 |
| tomcat | 14.1 | 2 | -1 | 3 |
| tradebeans | 45.7 | 3 | 2 | 5 |
| tradesoap | 95.0 | 2 | 0 | 5 |
| xalan | 20.9 | 4 | -2 | 3 |

Table 3.3: Average *Percent* Runtime Overhead for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against UNSAFESYNCMAP (convergence within 3%, OOM = Out of Memory)

| | | HASNEXT | | | UNSAFEITER | | |
|---|---|---|---|---|---|---|---|
| | ORIG (MB) | TM | Old | New | TM | Old | New |
| antlr | 4.3 | 4.4 | 4.1 | 3.8 | 4.8 | 4.0 | 4.5 |
| bloat | 4.9 | 40.3 | 19.3 | 13.9 | 7.8 | 175.4 | 79.0 |
| chart | 17.0 | 17.4 | 17.3 | 17.0 | 16.9 | 16.5 | 17.2 |
| hsqldb | 136.5 | 136.1 | 136.7 | 137.6 | 139.1 | 136.8 | 137.6 |
| jython | 4.9 | 5.1 | 4.7 | 4.8 | 5.5 | 5.1 | 5.0 |
| avrora | 4.7 | 4.6 | 12.1 | 9.1 | 4.4 | 114.0 | 15.8 |
| batik | 77.3 | 79.2 | 81.9 | 79.3 | 75.2 | 93.4 | 86.6 |
| eclipse | 101.0 | 100.8 | 104.0 | 97.1 | 98.3 | 100.3 | 110.3 |
| fop | 23.9 | 97.4 | 47.1 | 52.5 | 24.3 | 25.6 | 29.4 |
| h2 | 267.1 | 267.8 | 588.8 | 565.2 | 267.2 | 267.5 | 262.4 |
| luindex | 6.8 | 5.6 | 6.7 | 5.6 | 6.3 | 7.4 | 6.8 |
| lusearch | 4.6 | 4.7 | 4.6 | 4.8 | 4.6 | 4.3 | 4.2 |
| pmd | 22.3 | 56.9 | 65.5 | 48.5 | 17.2 | 147.2 | 86.4 |
| sunflow | 4.5 | 4.5 | 4.8 | 4.9 | 4.8 | 4.6 | 4.7 |
| tomcat | 11.7 | 11.4 | 11.6 | 11.4 | 12.5 | 11.8 | 11.5 |
| tradebeans | 62.9 | 62.9 | 62.4 | 62.1 | 63.7 | 63.9 | 64.1 |
| tradesoap | 63.9 | 61.8 | 64.8 | 63.3 | 63.4 | 64.7 | 64.4 |
| xalan | 4.9 | 4.9 | 5.0 | 5.1 | 4.9 | 5.0 | 4.9 |

Table 3.4: Peak memory usage (in MB) for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against HASNEXT and UNSAFEITER (during 5 iterations, OOM = Out of Memory)

| | | UNSAFEMAPITER | | | UNSAFESYNCCOLL | | |
|---|---|---|---|---|---|---|---|
| | ORIG (MB) | TM | Old | New | TM | Old | New |
| antlr | 4.3 | 4.1 | 4.0 | 4.6 | 4.1 | 4.2 | 4.2 |
| bloat | 4.9 | OOM | OOM | 56.7 | 6.7 | 100.0 | 48.3 |
| chart | 17.0 | 16.6 | 15.9 | 19.2 | 17.0 | 16.4 | 17.2 |
| hsqldb | 136.5 | 136.0 | 140.0 | 136.8 | 136.1 | 146.2 | 146.3 |
| jython | 4.9 | 6.1 | 20.9 | 5.1 | 5.3 | 4.9 | 5.4 |
| avrora | 4.7 | OOM | OOM | 8.5 | 4.3 | 18.4 | 12.6 |
| batik | 77.3 | OOM | 173.8 | 79.6 | 78.2 | 180.7 | 85.1 |
| eclipse | 101.0 | 106.9 | 198.9 | 101.1 | 100.4 | 115.1 | 90.1 |
| fop | 23.9 | OOM | OOM | 28.1 | OOM | OOM | 24.8 |
| h2 | 267.1 | 312.4 | OOM | 268.2 | 271.4 | 1456.7 | 265.5 |
| luindex | 6.8 | 7.4 | 6.8 | 6.9 | 7.4 | 7.5 | 7.5 |
| lusearch | 4.6 | 4.0 | 4.2 | 4.8 | 4.5 | 4.3 | 4.6 |
| pmd | 22.3 | OOM | OOM | 93.6 | 20.3 | OOM | 84.6 |
| sunflow | 4.5 | 4.7 | 4.6 | 4.4 | 5.1 | 4.4 | 4.9 |
| tomcat | 11.7 | 11.9 | 12.0 | 11.0 | 11.3 | 11.9 | 11.3 |
| tradebeans | 62.9 | 63.3 | 62.4 | 62.7 | 63.2 | 62.8 | 62.0 |
| tradesoap | 63.9 | 64.1 | 65.4 | 62.0 | 60.7 | 64.1 | 65.9 |
| xalan | 4.9 | 4.9 | 4.9 | 4.9 | 5.0 | 4.7 | 5.0 |

Table 3.5: Peak memory usage (in MB) for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against UNSAFEMAPITER and UNSAFESYNCCOLL (during 5 iterations, OOM = Out of Memory)

| | | UNSAFESYNCMAP | | |
|---|---|---|---|---|
| | ORIG (MB) | TM | Old | New |
| antlr | 4.3 | 4.6 | 4.4 | 4.9 |
| bloat | 4.9 | 6.9 | 25.8 | 12.3 |
| chart | 17.0 | 17.4 | 16.4 | 17.1 |
| hsqldb | 136.5 | 142.1 | 136.4 | 137.0 |
| jython | 4.9 | 5.8 | 5.0 | 5.1 |
| avrora | 4.7 | 4.4 | 12.4 | 4.9 |
| batik | 77.3 | 79.9 | 84.8 | 76.7 |
| eclipse | 101.0 | OOM | 102.3 | 98.7 |
| fop | 23.9 | OOM | OOM | 25.2 |
| h2 | 267.1 | 271.0 | 688.2 | 270.0 |
| luindex | 6.8 | 7.1 | 7.3 | 11.0 |
| lusearch | 4.6 | 4.6 | 4.8 | 4.7 |
| pmd | 22.3 | OOM | OOM | 32.9 |
| sunflow | 4.5 | 4.5 | 4.8 | 4.5 |
| tomcat | 11.7 | 11.4 | 11.3 | 11.8 |
| tradebeans | 62.9 | 64.0 | 62.7 | 64.0 |
| tradesoap | 63.9 | 65.5 | 65.1 | 65.6 |
| xalan | 4.9 | 5.1 | 5.1 | 4.9 |

Table 3.6: Peak memory usage (in MB) for Tracematches(TM), Previous JavaMOP(Old), and optimized JavaMOP(New) against UNSAFESYNCMAP (during 5 iterations, OOM = Out of Memory)

new version of JavaMOP shows 94% runtime overhead, while the previous version of JavaMOP shows 125% runtime overhead and hangs for three specifications, and Tracematches shows 1139% and hangs for two specifications.

Tables 3.4, 3.5, and 3.6 show the peak memory usage of the three systems. the new version of JavaMOP has lower peak memory usage than the previous version of JavaMOP in most cases. The cases where the new version of JavaMOP does not show lower peak memory usage are within the limits of expected memory jitter. However, memory usage of the new version of JavaMOP is still higher than the memory usage of Tracematches in some cases. Tracematches has several finite automata specific memory optimizations [10], which cannot be implemented in a formalism-independent system like JavaMOP. Although Tracematches is sometimes more memory efficient, it shows prohibitive runtime overhead monitoring bloat and pmd. There is a trade-off between memory usage and runtime overhead. The new version of JavaMOP uses a lazy garbage collection scheme [76] that does not immediately remove terminated monitor instances. If the new version of JavaMOP actively removed terminated monitors, memory usage will be lower, but at the cost of runtime performance. Overall, our monitor termination optimization achieves the most efficient—with respect to runtime overhead— parametric monitoring system with reasonable memory performance.

From these results, especially considering the fact that these cases are the worst combinations of benchmark programs and properties, we can conclude that our research on efficiency of runtime monitoring has been successful.

## 3.5 Chapter Related Work

Many approaches have been proposed to monitor program execution against formally specified properties (see the summary of related work in the Introduction to this thesis). Briefly, all runtime monitoring approaches except MOP have their specification formalisms hardwired, and few of them share the same logic.

There are four orthogonal attributes of a runtime monitoring system: logic, scope, running mode, and handlers. The logic specifies which formalism is used to specify the property. The scope determines where to check the property; it can be class invariant, global, interface, etc. The running mode denotes where the monitoring code runs; it can be inline (weaved into the code), online (operating at the same time as the program), outline (receiving events from the program remotely,

| Approach | Logic | Scope | Mode | Handler |
|---|---|---|---|---|
| JPaX [59] | LTL | class | offline | violation |
| TemporalRover [44] | MiTL | class | inline | violation |
| JavaMaC [82] | PastLTL | class | outline | violation |
| Hawk [41] | Eagle | global | inline | violation |
| RuleR [17] | RuleR | global | inline | violation |
| Tracematches [10] | Reg. Exp. | global | inline | validation |
| J-Lo [23] | LTL | global | inline | violation |
| Pal [29] | modified Blast | global | inline | validation |
| PQL [89] | PQL | global | inline | validation |
| PTQL [54] | SQL | global | outline | validation |

Figure 3.12: A Selection of Monitoring Systems

e.g., over a socket), or offline (checking logged event traces). The handlers specify what actions to perform under exceptional conditions; such conditions include violation and/or validation of the property. It is worth noting that for some logics, violation and validation are not complementary to each other, i.e., the violation of a formula does not always imply the validation of the negation of the formula. The Monitoring-Oriented Programming (MOP) framework allows for handlers for any number of user defined exceptional situations (called handler categories). Fig. 3.12 provides a brief breakdown of these systems. MOP, and, in particular, the JavaMOP instance of it presented in this chapter strives to be generic with respect to all of these attributes.

With respect to the coenable set based garbage collection scheme presented in this chapter, only the Tracematches system has studied the impact of garbage collection on parametric monitoring in any depth. As mentioned earlier, it uses a more precise formulation, which is similar, but based on the *state* of the monitor. Intuitively, the Tracematches garbage collection technique can be thought of as coenables sets indexed by state rather than events, but the formulation as presented in [10] is considerably different. While theirs is more precise, our empirical results, presented in Section 3.4.2, show that the coenable set technique is able to reduce memory usage in the JavaMOP system to comparable levels with Tracematches, while the JavaMOP system has considerably lower runtime overhead. More importantly, the Tracematches garbage collection technique is limited to finite logics, such as the regular expressions of Tracematches. However, our coenable approach is extensible to any underlying monitor implementation. We have a coenables sets generation algorithm for the context-free grammar plugin (see Chapter 6). A static state-based technique, such as the one used by Tracematches, could not be used for

context-free properties because the state space is unbounded.

Likewise, our suffix matching algorithm was inspired by the matching mode used in Tracematches, however, JavaMOP is able to select between total matching and suffix matching, and our algorithm is completely logical-formalism independent.

## 3.6 Chapter Conclusion

Efficient monitoring of parametric properties is a very challenging problem, due to the potentially huge number of parameter instances. Until now, solutions to this problem have either used a hardwired logical formalism, or limited their handling of parameters. Our approach, based on a general semantics of parametric traces with a property-based optimization, called enable sets, overcomes these limitations.

Additionally, We presented an effective novel garbage collection technique for monitoring parametric properties. Previous techniques were either completely agnostic to the property to monitor, thus incurring prohibitive runtime overheads due to memory leaks, or were intrinsically dependent on particular specification formalisms, thus being hard or impossible to use in other contexts. Our technique is the first which is both formalism-generic and efficient. As extensive evaluation shows, it is in fact significantly more efficient than the existing techniques, both formalism-generic and formalism-specific. Our results have at least two implications. On the one hand, they show that runtime monitoring of complex specifications can be used not only for testing, but also as an integral part of the deployed system. Indeed, in most practical cases the runtime overhead is negligible, so a well-designed recovery schema implemented by means of specification handlers can ensure highly dependable systems by simply not letting them go wrong at runtime. Note that the combinations program/property selected for evaluation in this paper were specifically chosen to be bad.

Our evaluation in this chapter shows that both of these optimization techniques are very important for improving the efficiency of Runtime Verification.

Two different techniques were demonstrated for improving the expressivity of formalism-independent parametric monitoring. Suffix matching allows for a method of pattern matching that is equivalent to that allowed by Tracematches, while the different parameter binding modes are completely unique to the representation used by JavaMOP, and allow for filtering undesirable results from monitors, such a property violations from monitor instances that bind unrelated variables.

# Chapter 4

# BusMOP

## 4.1 Chapter Introduction

This chapter presents all the research pertaining to BusMOP, which provides hardware implemented monitors for the purposes of monitoring bus traffic. It is implemented in two discrete instances: one for the PCI Bus in a typical computer, and an implementation that is used to enforce properties in a system on a chip (SoC) setting, presented as two separate case studies in this chapter.[1]

### 4.1.1 Chapter Contributions

This Chapter presents the first system to provide Runtime Verification capabilities for hardware and hardware/software co-designed platforms. By demonstrating its usefulness in both monitoring PCI bus traffic to determine misbehaving Commercial-Off-The-Shelf (COTS) components and its ability to make guarantees in a System-on-Chip (SoC) design platform, we show that Runtime Verification can be effective outside of the software application domain for which it was initially envisioned. Runtime Verification is particularly efficient in this domain, generally producing 0% overhead.

## 4.2 PCI Bus Monitoring

The real-time embedded system industry is progressively moving towards the use of Commercial-Off-The-Shelf (COTS) components in an attempt to reduce costs and time-to-market, even for highly critical systems like those deployed by the

[1]Work on BusMOP for the PCI bus is work with Rodolfo Pellizzoni, Marco Caccamo, and Grigore Roşu. It was originally presented in [100]. Work on BusMOP within a SoC context was performed with Rodolfo Pellizzoni, Min-Young Nam, Mu Sun, Marco Caccamo, and Lui Sha. It was originally presented in [101]. Note that syntax has changed since those papers in order to be more consistent with JavaMOP.

avionic industry. While specialized hardware and software solutions are sometimes available for such markets, their average performance and ease of integration is lagging behind the development of COTS components. For example, a commercial plane like the Boeing 777 uses the SAFEbus backplane [69], which, while specially designed to meet the hard real-time constraints of an avionic system, is only capable of transferring data up to 60 Mbps. On the other side, a modern COTS peripheral bus such as PCI Express 2.0 [98] can reach transfer speeds of 16 Gbyte/s, over three orders of magnitude greater than SAFEbus.

Unfortunately, when trying to use COTS for building high-integrity, real-time embedded systems, current engineering practices face significant challenges. While one can capture relevant assumptions about COTS as formal specifications, they are hard or impossible to formally verify: this is both because manufacturers are unwilling to disclose details of their implementation, for fear of losing competitive edge, and because the increase in performance is often matched by a similar increase in design complexity (out-of-order execution and branch prediction are examples of this trend in CPU design). Modern COTS peripherals running in master mode are particularly challenging. A master peripheral can directly communicate with all other elements in the system, including main memory and other peripherals, thus reducing the load on the CPU. On the other side, providing fault-containment becomes extremely hard: a misbehaving, low-criticality master peripheral could very well disrupt the entire system.

Based on the above discussion, our proposal for the safe integration of COTS peripherals in critical embedded systems is to use *runtime monitoring*: the peripheral requirement specifications are checked at runtime against its current observable behavior. If any violation is detected, then a suitable recovery action can be taken to restore the system to a safe state. The validity of the runtime monitoring approach has been proved in the field of software engineering by a large number of developed tools and techniques (see Section 4.5). However, applying runtime monitoring to our scenario poses some new challenges. First of all, the behavior of a COTS peripheral is controlled both by the hardware of the peripheral itself and by its software driver, hence we must check the correctness of their interactions. Second, master peripherals can directly interact with the rest of the system without requiring any action by the CPU. Based on these two considerations, our monitoring solution must be able to detect and check all communication between the peripheral and the rest of the system. Finally, runtime monitoring typically comes with an unforgivable price: runtime overhead. We can split such overhead

in two components: 1) overhead due to the observation and generation of relevant events 2) overhead due to running a monitor at each event to check if any property of the specification is violated. Both types of overhead tend to be unpredictable and thus unsuitable for real-time computation.

To combat these problems, we propose a distributed monitoring technique based on the development of a *monitoring device*. The idea is to introduce an additional hardware component into the system that can check all peripheral communication and perform recovery actions, when necessary. Assuming "sniffing" data transfers does not add delay to the system, our solution prevents the first type of overhead. The second type of overhead is removed by running all monitors directly on the device, adding no runtime overhead to the CPU. Additionally, the system can run completely undisturbed as long as no recovery action is needed.

The speed of modern COTS communication architectures rules out the possibility of a software implementation for the device; instead, all logic is implemented on a reconfigurable FPGA. Finally, to show that a monitored system is safe, we need to prove that the monitoring logic monitors, indeed, the right properties. In our system, this is ensured by automatically synthesizing the monitoring logic from formal requirements specification, so that it is "correct by construction". In particular, we leverage the Monitor Oriented Programming (MOP) framework (see Chapter 2), which is highly extensible and supports multiple formalisms, creating a new MOP instance: BusMOP.

**Illustrative Example.** An example of BusMOP can be seen in Fig. 4.1. This example is a property used in the case study of Section 4.2.4 and related to the behavior of Counter 2, a counter on the PCI703A board we used in our experiments. A complete formal description of the syntax used by BusMOP can be found in Chapter 2. This property, called SAFECOUNTERMODIFY, requires that any modification to cntr_cntrl2, the control register for Counter 2, happens only while the counter is not in use. This modification is captured by the cntrlMod event, because cntr_cntrl2 is at address X"220". The counter can be enabled/disabled by modifying bit 0 of cntr_cntrl2 (captured by the countEnable/countDisable events; "-" is the VHDL 'don't care").

Two monitor-local registers, cntrlCurrent and cntrlOld, are created and initialized to 0. These registers will hold the current and previous values of the cntr_cntrl2 register. This allows us to repair the register when/if the property is *violated* by writing the old value to the register on the peripheral itself (the value_reg assignment), and forcing the current value the monitor stores to be the previous value, as

```
pci SafeCounterModify{
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";

  event countDisable : memory write address = base1 + X"220"
                       dbyte value(0) in '0'
  event cntrlMod : memory write address in base1 + X"220"
                  {
                    cntrlOld <= cntrlCurrent;
                    cntrlCurrent <= value(15 downto 0);
                  }
  event countEnable : memory write address = base1 + X"220"
                      dbyte value(0) in '1'

  ere: ((countEnable countDisable) + cntrlMod + countDisable)*

  @validation {
    mem_reg <= '1';
    address_reg <= base1 + X"220";
    -- roll back to the previous cntr_cntrl2 value
    value_reg(15 downto 0) <= cntrlOld;
    cntrlCurrent <= cntrlOld;
    enable_reg <= "0011";
  }
}
```

Figure 4.1: Example Property: SAFECOUNTERMODIFY

can be seen in the violation handler section of the specification.

ere: before the property specification tells BusMOP that the property will be expressed using an extended regular expression pattern. The pattern itself matches any trace that consists of a cntr_cntrl2 modification, a disable of the counter, or an enable followed by a disable. The pattern is followed by *, allowing it to match repeatedly. The only way to violate this pattern, then, is to see a modification after an enable that is *not* followed by a disable first.

The implementation of the events, declarations, and the actions available to handlers is explained in Section 4.2.3. The formula/pattern implementation, and the use of handlers is explained in Section 4.2.3.

**Key contributions.** We provide three main contributions. First, in Section 4.2.2 we describe the design of a monitoring device for the PCI/PCI-X bus (a brief overview of PCI is presented in Section 4.2.1). The monitoring device can be plugged in on a PCI bus segment, and monitor all peripherals attached to the segment. Whenever peripheral activity fails to conform to the specification, the device can perform a *corrective* action: either bring the peripheral back to a safe state if the error is recoverable, or otherwise disconnect it from the system. While certain implementation decisions are necessarily specific to our choice of PCI, we believe that the general design principles and lessons learned can be applied to

most other communication architectures, as is illustrated in Section 4.3. Second, in Section 4.2.3 we provide a new instantiation of the MOP framework, called BusMOP, that is able to generate hardware modules; the generated monitoring logic is then integrated with the rest of the monitoring device design and synthesized on the FPGA. Third, in Section 4.2.4 we show the feasibility of the overall approach by applying our technique together with the developed monitoring device to check a COTS data acquisition board. Our experimental results reveal that the monitoring device is able to detect, and recover from, errors caused by faults in the driver that we discovered after manually inspecting it. We conclude by discussing related work in Section 4.5, and providing final remarks and future work in Section 4.6.

### 4.2.1 PCI Bus Overview

The Peripheral Component Interconnect (PCI) is the current standard family of communication architectures for motherboard/peripheral interconnection in the personal computer market; it is also widely popular in the embedded domain [98]. The standard can be divided in two parts: a *logical* specification, which details how the CPU configures and accesses peripherals through the system controller, and a *physical* specification, which details how peripherals are connected to and communicate with the motherboard. While the logical specification has remained largely unaltered since the introduction of the original PCI 1.0 standard in 1992, several different physical specifications have emerged since then.

One of the main features of the logical layer is plug-and-play (automatic configuration) functionality. On start-up, the OS executes a PCI base driver which reads information from special configuration registers implemented by each PCI-compliant peripheral and uses them to configure the system. Of peculiar importance is a set of up to 6 Base Access Registers (BARs). Each BAR represents a request by the peripheral for a block of addresses in either the I/O or memory space; the PCI base driver is responsible for accepting such requests, allocating address blocks and communicating back the chosen addresses to the peripheral, by writing in the BARs. To communicate with the peripheral, the CPU can, then, issue write and read commands, called *transactions*, to either I/O or memory space; each peripheral is required to implement *bus slave* logic, which decodes and responds to transactions targeting all address spaces allocated to the peripheral. Typically, address spaces are used to implement either registers, which control and determine the logical status of the peripheral, or data buffers. Peripherals *can* also implement

*bus master* logic: they can autonomously initiate read and write transactions to either main memory or the address space of another peripheral. Master mode is typically used by high-performance peripherals to perform a DMA transfer, i.e., transfer data from the peripheral to a buffer in main memory. The peripheral's driver can then read the data directly from memory, which is much faster than issuing a read transaction on the bus. Finally, each peripheral is provided with an interrupt line that can be used to send interrupts to the CPU.

There are two main flavors of physical architecture: PCI/PCI-X is parallel, while PCI-E is serial but runs at much higher frequency (2.5Ghz against up to 133Mhz for PCI-X). We have focused on PCI/PCI-X,[2] which implements a shared bus architecture. The logical PCI tree is physically divided into bus segments, and most bus wires are shared among all peripherals connected to a single segment. We refer to [98] for detailed bus specifications. Each transaction seen on the bus consists of an address phase, which provides the initial address in either memory or I/O space, followed by one or multiples data phases, each of which carries up to 32 or 64 bits of data for PCI/PCI-X, respectively (individual bytes can be masked using *byte enables*). Since each bus segment is shared, arbitration is required to determine which master peripheral is allowed to transmit at any one time. Arbitration uses two active-low, point-to-point wires between the peripheral and the bus segment arbiter, REQ# and GNT#. A standard request-grant handshake is used, where the peripheral first lowers REQ# to request access to the bus, and the arbiter grants permission to start a new transaction by lowering GNT#.

The entity that starts a transaction, either the CPU or a master peripheral, is known as the *initiator*, while the entity that receives the transaction, either a slave peripheral or main memory, is known as the *target*. All signals shown are active low. The peripheral first lowers REQ# to request access to the bus. After the arbiter grants access by lowering GNT#, the peripheral waits for the previous transaction to finish and then starts the address phase. During the address phase, the AD wires contain the address for the first data phase, while C/BE# determines the type of transaction: memory or I/O, read or write. During each data phase, the value is carried in AD and the address is implicitly incremented by 4/8 for a 32/64 bits bus respectively. C/BE# carries a set of byte enables for the value in AD; this permits to read/write only some of the bytes in AD in each data phase. The beginning and end of the transaction is signaled by the initiator using the FRAME# signal, while the

[2] We also plan to extend our design to PCI-E; see Section 4.6.

target uses DEVSEL# to signal that it has correctly decoded the address as being part of its address space. Finally, the IRDY# and TRDY# signals can be used by the initiator and target respectively to introduce wait cycles in the transaction: a data transfer happens only when both

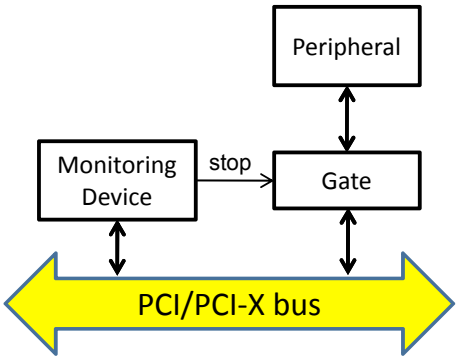
### 4.2.2 Monitoring Device

We designed a prototype monitoring device based on a Xilinx ML455 board [127] using a mixed VHDL/Verilog register transfer level (RTL) description. The board is outfitted with a Virtex-4 FPGA and is can be plugged into a standard 3.3Volts PCI/PCI-X socket. The FPGA implements both a slave and a master peripheral module, together with the monitoring modules. Events for the system are specified in terms of read/write data transfers on the bus and interrupt requests; the device continuously "sniffs" all ongoing activities on the bus, and it is therefore able to monitor communication for all other peripherals located on the same bus segment. Whenever a failure to meet the specification is detected, the device can execute a recovery action using strategies based on the detected error.

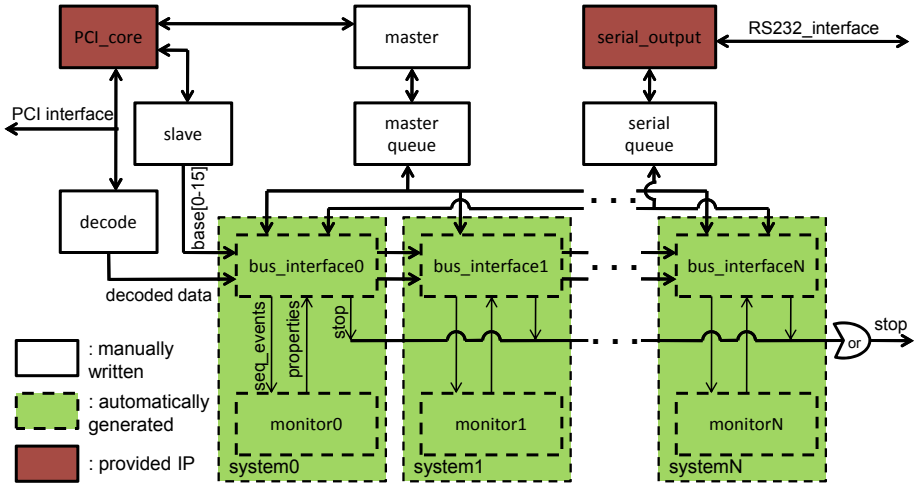
For a vast category of errors that involves incorrect interaction between the peripheral and its software driver, it is often possible to recover from the failure by forcing the peripheral into a consistent state. The monitoring device implements a master module, and can therefore initiate transactions on the bus. For example, consider a common type of error, where the driver fails to validate some input from the user and as a result writes an invalid value to a register in the peripheral. We can recover by rewriting the register with a valid value. However, if the error is caused by a fault in the peripheral hardware, interacting with registers may not be enough to bring the peripheral to a consistent and safe state.

We propose a mechanism that lets the monitoring device disconnect the faulty peripheral from the bus. We developed a simple hardware device, the *peripheral gate* [99], that is able to force the REQ# signal from the peripheral to the bus arbiter to be high; hence, the peripheral never receives the grant and it is prohibited from initiating any further transaction on the bus.[3] The peripheral gate is implemented based on a PCI extender card, i.e., a debug card that is interposed between the peripheral card and the bus and provides easy access to all signals. A clarifying

[3]While technically it is always possible for a faulty peripheral to disrupt the bus by altering the state of the signals, in practice the described approach is effective since access to the bus is mediated by three-state buffers enabled by GNT#.




(a) Gated Monitoring Device.




(b) Block Diagram.

Figure 4.2: Monitoring Device.

picture for monitoring of a single peripheral is provided in Fig. 4.2(a). The monitoring device can output a *stop* signal, which closes the gate when active high. Finally, sometimes the monitoring device cannot perform a suitable recovery action by itself, but there is a higher level actor, such as the OS or the system user, that can provide better recovery; examples include complex software operations such as restarting the driver or the whole PCI stack, and physically interacting with the peripheral. In this case, the best strategy is to communicate the failure to the chosen actor. The study of OS-level reliability techniques is outside the scope of this thesis; instead, for our prototype design we implemented a RS-232 controller that can be used to send information to the user over a serial connection.

The reader should notice that the nature of our implementation is such that if a trace is seen, which does not conform to a specification, as a consequence of a bus transaction, that specific bus transaction can not be prevented from propagating to the rest of the system. For example, if a faulty peripheral performs a write transaction to an area in main memory which is not supposed to modify, we can detect the error, disconnect the peripheral and report the failure to the OS/user. However, the information in the overwritten area will be lost. As part of our future work, we are working to implement an interposed monitoring device: by sitting between the bus and a peripheral, it will be able to buffer all transactions that target that specific peripheral or are initiated by it. If a property is validated/violated, it is then possible to take *preventive* measures (i.e., either discard or modify the transaction before propagating it). While this solution will provide a higher degree of reliability, there is a price to be paid in terms of increased communication delay due to buffering in the monitoring device.

A simplified block diagram for the monitoring device is shown in Fig. 4.2(b). We distinguish three types of blocks: 1) blocks provided by Xilinx as proprietary intellectual properties (IPs); 2) manually coded RTL modules provided by BusMOP, which are independent of the peripheral specification; 3) automatically generated RTL modules, which are dependent on the specification (see Section 4.2.3). PCI transaction signals are routed to two different modules: the PCI_core and the decode module.

The PCI_core module is a hard IP that implements all logic required to handle basic PCI functionalities such as plug-and-play. Bus slave and bus master logic is implemented by the slave and master modules, respectively. In particular, slave implements a set of 16 registers, base0 through base15. Since the OS configures the BAR registers at system boot, a peripheral cannot directly determine

the location of address blocks used by another peripheral. Hence, the OS must also write the locations of the address blocks allocated to monitored peripherals in the base registers. The decode module is used to simplify event generation. It translates all transactions on the bus (except for those initiated by the monitoring device itself) into a series of I/O or memory reads/writes, one for each data phase, as well as the occurrence of an interrupt, and forwards the translated information to the monitoring logic.

The system0, ..., systemI, ..., systemN blocks implement the monitoring logic for each of $N$ user specified properties. Each systemI block consists of two automatically generated modules: bus_interfaceI contains all logic that depends on the specific choice of communication interface (PCI bus), while monitorI contains all logic that depends on the formal language used to specify the property. This separation provides good modularity and facilitates code reuse. bus_interfaceI first receives as input the decoded bus signals and generates events, which are sequentialized by the events_sequentializer submodule (see Section 4.2.3), and then passed to monitorI using the seq_events wires. monitorI checks whenever the formula for the I-th property is validated/violated and passes the information back to bus_interfaceI, which can then execute three types of recovery: 1) disconnect a monitored peripheral from the bus using the stop signal; 2) send information to the user using the serial_output module, which implements a RS-232 transmitter; 3) start a write transaction on the bus using the master module. Finally, since it is possible for multiple systemI modules to initiate recovery at the same time, we provide queuing functionalities for serial_output and master in modules master_queue and serial_queue, respectively.

It is important to notice that in the current implementation the time elapsed from any event that triggers a validation/violation to executing the corresponding handler is at most 4 clock cycles. This time is short enough to execute a recovery action before a faulty peripheral is allowed to start a new transaction, as PCI arbitration overhead prevents a peripheral from transmitting immediately.

### 4.2.3 Property Specifications

Properties are specified using a domain specific event syntax, and formulae or patterns written in the logic of a particular plugin. Additional monitor state can also be declared using the declarations section. The violation handler and validation handler sections allow for arbitrary code to be executed on the occurrence of a

violation or validation, respectively. An example of how they are used can be seen in Fig. 4.1 in Section 4.1. Currently, we have support for the extended regular expression (ERE) and past-time temporal logic (PTLTL) MOP Plugins, and adding most of the others will require a minimal amount of work, as only the monitor component changes from one logical specification formalism to another. This means that properties may be specified, formally, using an ERE pattern or a PTLTL formula. BusMOP was created before the MOP logic repository was in its current form, and will be eventually updated to use it. At that time it will support all finite-state logical formalisms.

### Events

A formal description of BusMOP event syntax can be found in Chapter 2, as well as BusMOP syntax in general. There are three basic types of events in BusMOP: I/O accesses, memory accesses, and interrupts. It is important to distinguish between I/O and memory events because they require different enable functionality and different read/write signals. I/O and memory events must specify at least an address, which may be an arithmetic expression over identifiers, VHDL numbers, addition, subtraction, and concatenation, and whether the event is a read or a write. An I/O or memory event may also specify a value range, which is the value of the address read or written by the bus transaction. Ranges can consist of a single arithmetic expression, or a pair of comma separated arithmetic expression denoting the minimum and maximum values that may trigger the event (thus, ranges are inclusive). Value ranges must also specify a size, byte, dbyte (16 bits), or qbyte (32 bits), so that the correct comparison code and byte enables can be generated (values smaller than a byte require masking the proper bits). Address ranges are used in events that *do not* have specified value ranges. The reason for this is that when a value range is specified, the code generator must generate the proper byte enables based on address alignment, and alignment does not make sense for ranges. Address ranges are useful for some properties, e.g. a property that monitors accesses to a certain buffer in memory.

### The bus_interface Module

The code for all declarations handlers is copied verbatim into the VHDL module defining the bus_interface. Because of this copying, the code must be written in VHDL. The events are expanded to combinatoric statements implementing

the specified logic. The output of the combinatoric statements is assigned to an events wire vector, which is connected to the monitor module through an event_sequentializer submodule. Each index in the bus corresponds to the truth value of a specific event, numbered with the 0'th index as the first event, and the n'th index as the n'th event from top to bottom in the specification. This ordering is important, because it directs the event linearization performed by the event_sequentializer submodule.

The event_sequentializer is necessary because the logical formalisms expect linear, disjoint events. The event_sequentializer takes coincident events and sends them to the monitor in subsequent clock cycles, in ascending index order, using the seq_events wire vector. Therefore, if events(0) and events(3) occur in the same cycle, the monitor will see 0 followed by 3. To see why simultaneous events are possible, consider, again, Fig. 4.1 from Section 4.1. The cntrlMod event is asserted whenever the cntr_cntrl2 register (base1 + X"220") is written. Because both the countEnable and countDisable events require writes to the same address as the cntrlMod event, any time countEnable or countDisable are triggered, a cntrlMod is also triggered. As the property tries to enforce the policy that all modifications happen when the counter is not enabled, we must serialize events such that cntrlMod happens *after* a countDisable and *before* a countEnable. The ordering of events in Fig. 4.1, is consistent with this, because countDisable is listed before cntrlMod, which is listed before countDisable.

The handler is placed in the module such that it is only executed if the monitor module denotes that the property has been violated. The situation is similar for a validation handler, save that it is executed only when the formula or pattern is validated. As can be seen in the Fig. 4.2(b), the monitor module reports the validation, violation, or neutral state of the monitored property, via the properties wire vector, to the bus_interface module. Several actions are available in validation and violation handlers. Aside from manipulating any local state of the monitor (such as the write to cntrlCurrent in Fig. 4.1), the bus_interface module makes available several registers which can be used used to execute the recovery actions detailed in Section 4.2.2. The registers are summarized in the table below for convenience, they may also be found in Chapter 2:

| Write Interface | |
|---|---|
| io_reg | write request in I/O space |
| mem_reg | write request in memory space |
| address_reg | write address |
| value_reg | write value |
| enable_reg | byte enables |
| serial_reg | ASCII value to serial output |
| stop_reg | Peripheral gate control |

As can be seen in Fig. 4.1, we perform a memory write to the cntr_cntrl2 register of its previous value. The address_reg is used to denote the address of cntr_cntrl2 (base1 + X"202"), while the value_reg is set to the old value of cntr_cntrl2, the mem_reg is asserted to tell the PCI bus that the write performed is a memory write, and the byte enables are set to "0011" to denote that the lower two bytes must be written.

#### The monitor Module

The monitor module is responsible for monitoring the property given serialized events. It encompasses the logic of the formula, and it is the only portion of our system dependent on the logical formalism used. More information on the ERE and PTLTL plugins used by BusMOP can be found in Chapter 5.

A design decision relating to both logics we have implemented, and all future logics, is that properties cannot be violated or validated before an event arrives. This helps eliminate some strange interactions. JavaMOP has the same functionality, but in JavaMOP it is due to the fact that a monitor does not *exist* before the first event, whereas in BusMOP, the monitor exists as soon as the FPGA is configured.

### 4.2.4 Case Study: The PCI703A ADC/DAC Board

In this section, we show how our runtime monitoring technique can be applied to a concrete case by providing specification and runtime experiments for a specific COTS peripheral, the PCI703A board [47]. PCI703A is a high performance Analog-to-Digital/Digital-to-Analog Conversion (ADC/DAC) peripheral for the PCI bus. In particular, it can perform high-speed, 14-bits precision ADC at a rate of up to 450,000 conversions/s, and transfer data to main memory in bus master mode. At

the same time, the peripheral is simple enough that we were able to carefully check all provided hardware manuals and to manually inspect its Linux driver; specifying formal properties for a peripheral clearly requires a deep understanding of its inner working. In our proposed model, the peripheral's manufacturer is responsible for writing the runtime specification. In this sense, the formal specification can be thought of as a correctness certification provided by the manufacturer, as long as the user employs a monitoring device and recovery actions can be proved to restore the system to a safe state.

To better mimic what we think would be a typical process for a COTS manufacturer, we produced a requirement specification for the PCI703A in two steps. First, we prepared a detailed description of the communication behavior of the peripheral in plain English. Then, we converted this informal description into a formal set of events and formulae for BusMOP. Inspection of the driver revealed two software faults, both of which can cause errors that are detected and recovered by the monitoring device. While in this case we could have prevented errors by simply removing the faults, we argue that drivers for more complex peripherals can be thousands of lines long and neither code inspection nor testing is sufficient to remove all bugs. We further injected additional faults in the driver to test all written formal properties. It would have been nice to also show recovery for hardware faults, but we did not find any in the tested peripheral and injecting faults in the hardware is difficult. In what follows, we first provide an overview of PCI703A and then we detail properties for an example subsystem, a counter used in the ADC process. The example is particularly instructive as we show how a small but representative set of properties is able to catch one of the aforementioned driver bug.

A block diagram for the PCI703A is shown in Fig. 4.2.4. The bus slave logic implements two memory address blocks in BAR0 and BAR1, used for conversion data and control registers, respectively; the corresponding base addresses are written in base0 and base1 in the monitoring device. The ADC Control and DAC Control blocks control the ADC/DAC operations and write/read data into internal FIFOs. The DMA Control block can be programmed to move data between each FIFO and main memory using bus master functionality. Finally, the Counter Timers block implements four counters. Counter 0 and 1 are user programmable and can be used either for debugging purposes or to trigger a DA conversion. Counter 3 is also user programmable and produces an external output. Finally, Counter 2 is not meant to be user programmable; it is to be used exclusively to generate the clock for AD conversions. The C user library provided with the driver exports

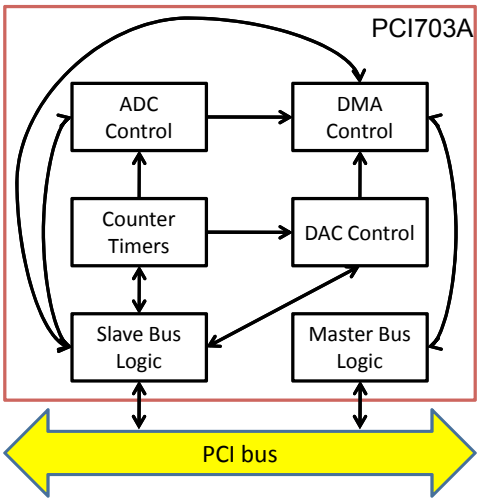



Figure 4.3: PCI703A Diagram.

an ADConfig function used to configure ADC Control and the associated Counter 2. The library also provides a CTConfig function to be used to configure the user counters; unfortunately, under Linux the function can also be used to change the configuration of Counter 2. This is a problem, as any user in the system could erroneously or maliciously change Counter 2 while an ADC is in progress.

Three 16-bits control registers are relevant to our discussion: cntr_cntrl2 (at hexadecimal location 220 relative to BAR1), cntr_divr2 (228), and adc_cntrl (300). Bit 0 of cntr_cntrl2 determines whether Counter 2 is enabled, and bits 2-1 determine its clock source (either 20Mhz or 100Khz); when the counter is enabled, it first loads the content of cntr_divr2 and then starts counting down at the selected frequency. When it reaches zero, the value of cntr_divr2 is reloaded, a clock signal is sent to ADC Control, and finally if bit 4 of cntr_cntrl2 is set, an interrupt is generated. Register adc_cntrl controls the behavior of ADC Control; in particular, bit 0 enables/disables the ADC process and bits 2-1 determine the clock source, with a value of "00" indicating that Counter 2 is used.

We express three requirements:

**Requirement 1** *Bit 4 of cntr_cntrl2 should never be set. While the functionality is relevant for Counters 0,1, in the case of Counter 2 setting bit 4 would cause the generation of spurious interrupts that increase load on the driver.*

**Requirement 2** *If the ADC is using Counter 2, and the clock source for Counter 2 is set to 20 Mhz, then the value of cntr_divr2 must be at least 45 to avoid violating*

```
pci InterruptFix{
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";

  event cntrlMod : memory write address in base1 + X"220"
                   {
                     cntrlOld <= cntrlCurrent;
                     cntrlCurrent <= value(15 downto 0);
                   }
  event setBit4 : memory write
                  address = base1 + X"220"
                  dbyte value(4) in '1'

  ere: setBit4

  @match {
    mem_reg <= '1';
    address_reg <= base1 + X"220";
    -- roll back to the previous cntr_cntrl2 value
    value_reg(15 downto 0) <= cntrlOld;
    cntrlCurrent <= cntrlOld;
    enable_reg <= "0011";
  }
}
```

Figure 4.4: INTERRUPTFIX Specification

*the maximum conversion speed of the peripheral.*

**Requirement 3** *If the ADC is active and using Counter 2, then Counter 2 must also be active; furthermore, while Counter 2 is active no change to the counter configuration is allowed.*

Requirements 1-3 are able to catch the driver bug in the sense that an invalid counter configuration can not be set before starting the ADC, and furthermore while the ADC is active no counter modification is allowed. We wrote four (five including the example from Section 4.1) formal properties to capture the requirements:

INTERRUPTFIX**.** The INTERRUPTFIX specification is the formalization of Requirement 1, and can be seen in Fig. 4.4. Because we do not want the 4th bit set, we simply monitor the pattern setBit4, an event which corresponds to setting the 4th bit. We perform recovery when the pattern is validated by overwriting cntr_cntrl2 with the last valid value, similarly to SAFECOUNTERMODIFY in Fig. 4.1.

SAFECONVERSIONSPEED**.** The SAFECONVERSIONSPEED specification is the formalization of Requirement 2, and can be seen in Fig. 4.5. For this property we chose to show how event side effects can be used in handlers as part of checking that a property has been validated/violated. When the clkSrcSet or srcSet events

```
pci SafeConversionSpeed{
  signal clkSrc : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal src : STD_LOGIC_VECTOR(15 downto 0) := X"0000";

  event divrBad: memory write address = base1 + X"228"
                 dbyte value in 0,44
  event divrGood: memory write address = base1 + X"228"
                  dbyte value in 45,65535
  event clkSrcSet : memory write address in base1 + X"300"
                    { clkSrc <= value(15 downto 0); }
  event srcSet :  memory write address in base1 + X"220"
                  { src <= value(15 downto 0); }
  event countEnable : memory write address = base1 + X"220"
                      dbyte value(0) in '1'

  ere : (divrBad (clkSrcSet + srcSet)* countEnable)*

  @match {
    if (clkSrc(2 downto 1) = "01") and (src(2 downto 1) = "00") then
      mem_reg <= '1';
      address_reg <= base1 + X"228";
      --set cntr_divr2 to 45
      value_reg(15 downto 0) <= X"2D";
      enable_reg <= "0011";
    end if;
  }
}
```

Figure 4.5: SAFECONVERSIONSPEED Specification

are triggered, meaning that the cntr_cntrl2 or adc_cntrl registers have been modified, respectively, we store the value written to the register in monitor local registers (e.g., src <= value(15 downto 0)). The pattern specifies that the cntr_divr2 be set to a bad value (less than 45), followed by any number of updates to cntr_cntrl2 or adc_cntrl, followed by the enabling of the counter. If cntr_divr2 is set to a value larger than 44, the pattern will be violated, and the monitor will be reset. This means that the validation handler will be executed only when then value of cntr_divr2 is too low for safe conversion, but regardless of whether or not the board is actually using Counter 2. The handler then checks that it is, in fact using Counter 2, and that Counter 2 is using the 20Mhz source, before performing the recovery: setting cntr_divr2 to a valid value (45).

NODISABLEWHILECONVERTING**.** The NODISABLEWHILECONVERTING specification is the formalization of part of Requirement 3, and can be seen in Fig. 4.6. This could have been written in a similar manner to SAFECONVERSIONSPEED, i.e., using event side effects to store current register values and checking them in the handler. We decided to use a fully formal specification, that defines events for setting the registers to good or bad values. The formula itself specifies that, if the ADC is enabled, and clkSrc2 is good, meaning that Counter 2 is being used

```
pci NoDisableWhileConverting{
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";

  event countEnable : memory write address = base1 + X"220"
                      dbyte value(0) in '1'
                      {
                        cntrlOld <= cntrlCurrent;
                        cntrlCurrent <= value(15 downto 0);
                      }
  event countDisable : memory write address = base1 + X"220"
                       dbyte value(0) in '0'
                       {
                         cntrlOld <= cntrlCurrent;
                         cntrlCurrent <= value(15 downto 0);
                       }
  event clkSrc2Good : memory write address = base1 + X"300"
                      dbyte value(2 downto 1) in "01"
  event clkSrc2Bad : memory write address = base1 + X"300"
                     dbyte value(2 downto 1) not in "01"
  event adcEnable : memory write address = base1 + X"300"
                    dbyte value(0) in '1'
  event adcDisable : memory write address = base1 + X"300"
                     dbyte value(0) in '0'

  ptltl :  ( ((not adcDisable) S adcEnable) and
             ((not clkSrc2Bad) S clkSrc2Good) )
           implies
           ((not countDisable) S countEnable)

  @violation {
    mem_reg <= '1';
    address_reg <= base1 + X"220";
    -- roll back to the previous cntr_cntrl2 value
    value_reg(15 downto 0) <= cntrlOld;
    cntrlCurrent <= cntrlOld;
    enable_reg <= "0011";
  }
}
```

Figure 4.6: NODISABLEWHILECONVERTING Specification

to time the ADC, then Counter 2 must be enabled. The part of the formula before the implies keyword, states that the ADC is enabled and the ADC clock source is Counter 2, the second half of the formula is the requirement that Counter 2 not be disabled. The formula is true when correct behavior is exhibited, so we use a violation handler for the recovery action, which again is simply to set cntr_cntrl2 to the last valid value.

SAFEDIVRMODIFY**.** The SAFEDIVRMODIFY specification is the formalization of part of Requirement 3, and can be seen in Fig. 4.7. In conjunction with NODISABLEWHILECONVERTING and SAFECOUNTERMODIFY (from Section 4.1), all of requirement 3 is covered. This specification ensures that cntr_divr2 is not modified while Counter 2 is enabled. This property is the same as SAFE-

```
pci SafeDivrModify{
  signal divrCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal divrOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";

  event countDisable : memory write address = base1 + X"220"
                       dbyte value(0) in '0'
  event divrMod : memory write address in base1 + X"228"
                  {
                    divrOld <= divrCurrent;
                    divrCurrent <= value(15 downto 0);
                  }
  event countEnable : memory write address = base1 + X"220"
                      dbyte value(0) in '1'

  ptltl: (divrMod) and (*)((not countDisable) S countEnable)

  @validation {
    mem_reg <= '1';
    address_reg <= base1 + X"228";
    -- roll back to the previous cntr_divr2 value
    value_reg(15 downto 0) <= divrOld;
    divrCurrent <= divrOld;
    enable_reg <= "0011";
  }
}
```

Figure 4.7: SAFEDIVRMODIFY Specification

COUNTERMODIFY from Fig. 4.1, save that we are ensuring that cntr_divr2 is not modified, rather than cntr_cntrl2. We also used PTLTL rather than ERE, to show how two very similar properties look in both logics. These could be collapsed into one specification, but it would make recovery more complicated, because we only want to roll back the register that was actually modified (cntr_cntrl2 or cntr_divr2). The formula itself states that if cntr_divr2 has been modified and the counter has not been disabled since the last time it was enabled, than we must recover. Unlike SAFECOUNTERMODIFY we use a validation rather than a violation handler, because the formula was easier to express with recovery being on validation.

As a final consideration, note that the handlers of SAFECOUNTERMODIFY, INTERRUPTFIX and NODISABLEWHILECONVERTING can be invoked simultaneously if an incorrect value is written to cntr_cntrl2, which results in the execution of multiple bus writes. However, this causes no problem since all handlers overwrite cntr_cntrl2 with the same valid value.

## 4.3 SoC Monitoring

Systems-on-Chip (SoCs) are increasingly popular in the embedded system domain because they consume less power and cost less money than the multi-chip solutions

they replace. We believe that the SoC paradigm is especially promising for safety-critical embedded systems such as those employed in the medical market [126]: it is easier to formally verify the correctness of critical functionalities when they are implemented in hardware rather than in software on top of an OS and library stack [11]. Furthermore, SoC design can offer higher hardware reliability.

However, implementing safety-critical applications on SoC brings additional design challenges. The high degree of integration possible in modern fabrication processes can easily lead to *mixed-criticality systems*: multiple applications with different criticalities run simultaneously on the same chip. As an example, consider a medical pacemaker (a more detailed description can be found in Section 4.3). Implanted cardiac pacemakers are used to provide pacing assistance for patients with slow or abnormal heart rates. The life-critical pacing component uses leads to send shocks to the heart to force contraction. However, modern pacemakers also implement a variety of other applications such as data logging and remote communication and programming that are used for diagnosis. Since these lower-criticality functionalities must communicate with the life-critical module and share physical resources such as battery energy, memory and communication bandwidth, faults could potentially propagate from a lower to a higher criticality application. This is a daunting problem because formally verifying and/or certifying non safety-critical modules is either extremely expensive or unfeasible: they typically make large use of both software and hardware Intellectual Properties (IP) such as CPU cores and OSes that are very complex. This is a domain in which BusMOP can play a crucial part to providing safety and reliability.

In this section, we introduce a new design methodology for SoC that specifically targets mixed-criticality systems. Our methodology, which relies heavily on BusMOP, provides strong isolation guarantees to applications and imposes limits to fault propagation. In detail, we distinguish between functional isolation, which deals with the correct exchange of data, and physical isolation, which deals with the correct sharing of system resources. Our methodology is similar to the concept of *Platform-Based Design* (PBD) [107].

A platform is a library of components. A platform instance is a set of library components selected to generate a concrete design. In a PBD design flow, the designer first specifies the system functionality using an implementation-independent description language. The designer then selects a suitable platform and performs *mapping* of functional elements to platform components, thus creating a platform instance.

Our platform contains three types of architectural components: processors (which can either be CPUs or *hardware processors*, e.g. logic circuits implementing a specific functionality), communication infrastructures and memories. Functional specification, architectural specification and mapping are all performed using the Architectural Analysis and Design Language (AADL) [50]. AADL is rapidly gaining support in safety-critical markets such as avionic and medical domain and it has been applied to many industrial examples (see Section 4.3). Mapping is performed by binding functional processes to architectural processors and specifying processes' requirements in term of data transfers and resource usage. This requirement specification creates a *certificate* for each process, e.g. it determines the complete set of acceptable runtime behaviors for the process.

We provide isolation using the following key idea. Each processor is encapsulated in a *monitoring hardware wrapper* that leverages the power of BusMOP to control all communication and resource usage by the processes executed on that processor, and can therefore enforce their certificates at runtime. System level verification is greatly simplified because it is sufficient to verify that the system is correct based on the certificates of low-criticality applications, and not the complex behavior of the IPs that implement them.[4]

To summarize, our main contributions are as follows. 1) We developed a new PBD methodology for SoC based on AADL that is targeted for safety-critical systems. In particular, we developed tools that are able to automatically generate monitoring wrappers based on an AADL specification. 2) We employ BusMOP to enforce functional isolation by formally checking the communication behavior of processes at runtime. If a violation is detected, the wrapper is able to take a *preventive* measure by rejecting the faulty communication. 3) At the physical isolation level, we derive a static coschedule of computation and communication and assign timing reservations to all processes in the form of temporal budgets based on their certificates. The wrapper checks usage of both computation and communication resources and prevents a process from exceeding its assigned budget. Finally, in battery-operated devices the platform can monitor overall power consumption and use the wrappers to shut down low-criticality processes and save energy for higher criticality applications.

[4]Note that obviously the platform can not prevent a low-criticality process from failing if the unverified processor on which it is executed suffers a critical internal error. Therefore, the certificate must state that such process can stop at any time. However, the platform will prevent the internal error from propagating to other processors.

Our AADL-based system modeling is introduced in Section 4.3, together with a case study based on a medical pacemaker; we continue to elaborate on the case study in the remainder of this chapter. Section 4.3.2 describes our architectural implementation, focusing on the wrappers. Sections 4.4 details functional isolation. Timing isolation is detailed in [101]. It is not detailed here because it is not related to BusMOP.

### System Model

The importance of model-based Architecture Description Languages (ADLs) and formal analysis of system models is becoming apparent in the industry of hard real-time systems, such as avionics, aerospace, and medical devices. Model-based ADLs provide the automation capability of generating beneficial abstractions from system models to be verified or examined by computed-aided tools. AADL is a type of ADL initially developed for the avionic market. It is based on 15 years of research, including the MetaH language developed by Honeywell Labs and several DARPA programs [20]. AADL-based tools and analysis methodologies have been developed in the area of safety analysis [119], dependability [106] and schedulability [72, 113, 116]; however, to the best of our knowledge this is the first time that an AADL-based model is used to automatically generate code that enforces safe process behavior. In this section, we first describe how AADL can be used to provide both functional specification and mapping in our platform. Then, we introduce our case study of a medical pacemaker.

**AADL-Based Modeling.** AADL lets the designer specify the logical functional model separately from the hardware platform. An AADL specification is comprised of different components and their interactions. Components used for the logical design include process, thread, and data. AADL execution platform components include processor, memory and bus; they have a one-to-one correspondence with our architectural components. Each processor represents either a custom-built hardware processor or a CPU. Memory is used, among others, for all external memories (such as external SRAM or DRAM chips) used to hold shared data. Bus represents the unique communication infrastructure used to interconnect all processors and memories. Finally, the system component is a special composite component which is used to either encapsulate other components, divide them into groups or represent any new abstract entity that can not be modeled by preexisting components. In our design methodology, the logical components and architectural

components belonging to a platform instance are grouped in separate system components for modularity and clarity of representation. All components are tagged with properties which add extra information that can not be expressed by structural descriptions (for example, the processor type is specified by a class property). The core AADL standard has pre-declared properties that support real-time scheduling as well as other areas of research. AADL also provides the syntax to add new user-defined properties.

Every application is modeled as a collection of processes with one or more thread subcomponents. Threads are active agents: they receive inputs, process and output data. Period, deadline, and execution time properties are associated with each process as a timing reservation for all of its subcomponent threads. The platform guarantees that at each period a process is provided with an amount of execution time at least equal to its request within a time window equal to the specified deadline. Each process represents a dedicated memory and design space protected by the system architecture. The platform ensures that all threads inside a process are functionally isolated from other threads running on different processes.

Two types of inter-process communication are supported. A process can declare any number of input *message queues* to which other processes can send data. Furthermore, shared data objects can be declared and accessed by any process. In AADL, message queues are modeled as event data port connections and data accesses are used to connect processes to shared data object. Each communication has an associated data component which represents the type and size of data that is to be sent or shared and an associated deadline property which represents the maximum allowed amount of time to complete the data transfer. The acceptable communication behavior of a process can be further specified by a PropertyList and associated EventLists. Each entry in an EventList describes a specific communication event, e.g. a send/receive to a message queue or a DMA read/write to a data object. Properties are defined using BusMOP. Recall that property syntax is described in Chapter 2.

Platform mapping is performed by "binding" logical components to architectural components. Each process is bound to a processor and each data component is bound to a memory (either a message queue or an external memory). Each HW processor can execute a single process, while a CPU processor can execute multiple processes. Note that some properties of logical components depend on the binding. For example, required execution time depends on the processor to which a process is bound. The values of these properties need to be re-examined every

time a related change in the mapping occurs.

The AADL platform specification would not be nearly as useful if the designer had to manually check it for correctness and convert it into an implementation, since the potential for human error during translation is very high. As such, we built a set of tools to assist the design flow. Our AADL tool makes sure that the model conforms to predefined *analysis specific* rules for correctness; these include all aforementioned binding rules. The tool also automatically derives some property values required for implementation which depend on the binding (in particular, addresses are associated with all memory elements), and generates an XML file based on a customized schema containing a description of the platform instance. In particular, each process is characterized by a certificate specifying timing requirements for the process itself and its communication plus all attached formal properties. Additional tools read the XML file and automatically generate the monitoring wrapper; a detailed description is provided in Sections 4.3.2, 4.4.

### 4.3.1 Case Study: Medical Pacemaker

Pacemakers are one of the most critical medical devices [126]. Once they are implanted, they must continue to operate correctly with very little maintenance for at least 5 years. Having methods of self-diagnosis and recovery of software and hardware errors could go a long way for prolonging the lifespan of reliable operation. There have been designs to ensure the safety of pacemaker systems in terms of the control algorithm [11], including model-checking of the pacing controller using UPPAAL [122]. However, such previous work is based on the assumption that the underlying implementation can offer strong isolation guarantees for memory, power and communication. Here, we take a more general approach where isolation guarantees are not a by-product of the system implementation, but are instead specified as requirements in the functional model.

A typical functional decomposition of cardiac pacemaker is shown in Fig. 4.8. A pacemaker is connected to the heart via leads (typically two). These leads provide EKG signals which the internal pacer uses to detect whether the heart is contracting properly and to send shocks to the heart to force contraction. The internal pacer has various control parameters which can be tuned to each patient: these include a reference rate of pacing, the operational mode, and the thresholds for sensing and actuation. The operational mode determines how to pace the heart; modes can determine which leads are used for sensing if any, whether rate adaptation is used,

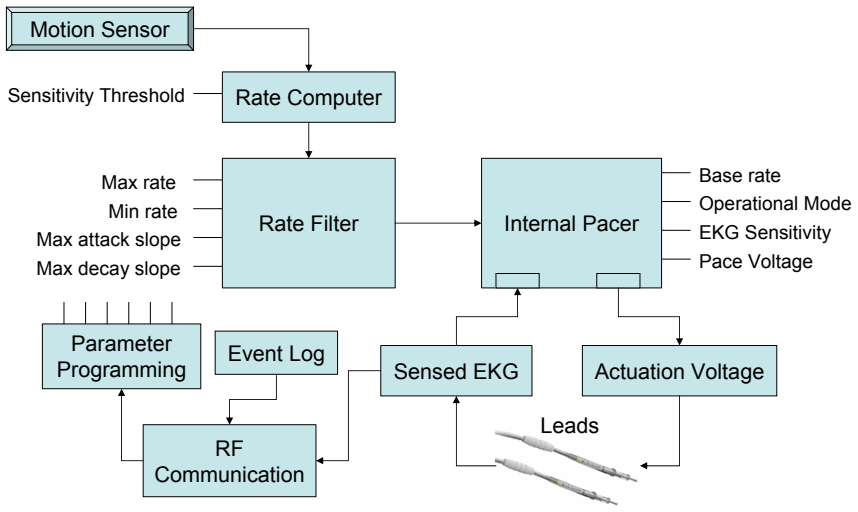



Figure 4.8: Pacemaker Block Diagram

and whether certain types of special therapies should be provided.

A simple pacemaker only needs the leads and the internal pacer to be functional. However, for more effective pacing, it is desired to have the pacing rate adapt to the activity levels of the patient. Rate adaptation normally uses some form of motion sensing, usually through accelerometers or pressure sensors. The motion data is then used to compute a rate. A rate computation uses a sensitivity threshold to filter out motion noise (i.e. when riding a car). Furthermore, to match biological characteristics, the computed heart rates values are bounded (min and max rate) as well as their rates of change (attack and decay slope).

Finally, the pacemaker employs a communication component using RF signals. RF communication allows medical personnel to program various parameters of the pacemaker without intrusively removing the pacemaker. Furthermore, logged events and real time sensor data can be sent to the medical personnel during diagnostics and check-up.

A schematic of an AADL-modeled pacemaker platform instance is shown in Fig. 4.9. The life-critical functionality is the core pacing implemented by the *Pacer* process in a HW processor. The Pacer includes the sensor and actuation interfaces to the leads and also the pacing logic. *EKG Sensor* data arrives at the frequency of 128 16-bit samples per second, which also determines the Pacer period. Also, the Pacer logs *EKG Signal* values and *Shock Event*s in the signalBuffer memory area for later retrieval and diagnosis.

The values written to the signalBuffer are time-stamped and will eventually be

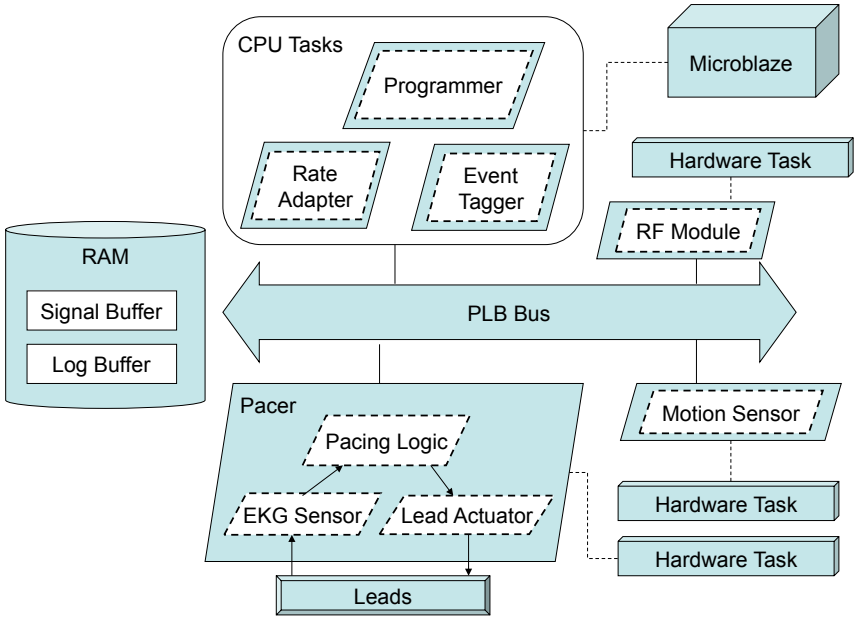
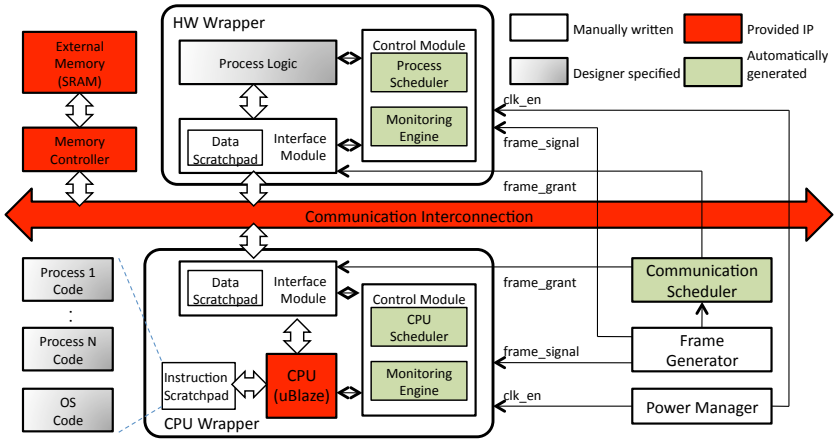


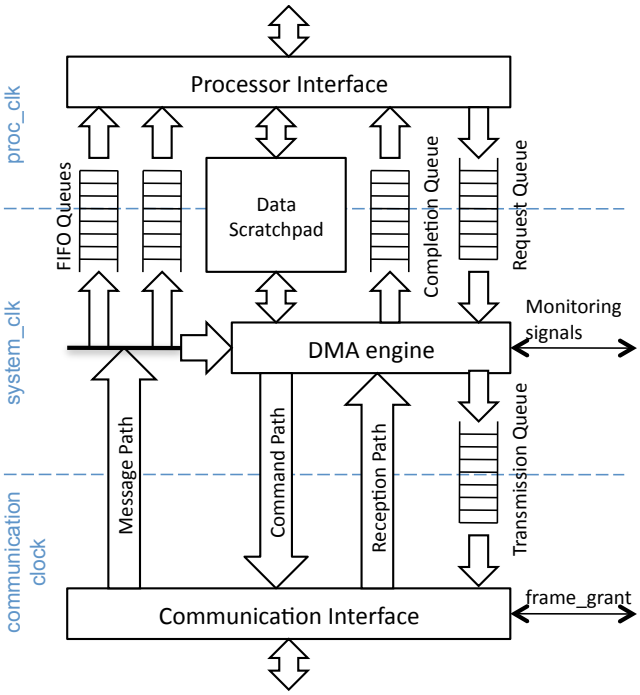

Figure 4.9: Mapped Pacemaker Platform

overwritten in a circular manner. The *Motion Sensor* process is also implemented in hardware. It samples data at 80Hz and sends the measured values to the *Rate Adapter* through a message queue.

Three processes requiring complex microprocessing functionality are executed on a Microblaze CPU. The Rate Adapter uses motion data to compute a reasonable rate required for pacing and sends it to the Pacer twice per second; its failure is not life-critical but could still cause significant discomfort to the patient. The other two processes are less critical since they do not immediately affect the condition of the patient. The *Event Tagger*, running at 8Hz, reads in a window of EKG signal values and shock events from the signalBuffer and finds anomalies that are then logged to another section of memory (the logBuffer) for more permanent storage and later diagnostics. Finally the *Programmer* is used to send parameter updates. It processes commands sent by medical personal through the *RF Module* (implemented in hardware) and it sends rate adaptation parameters (ie. max/min rate, attack/decay slope) to the Rate Adapter process and pacing parameters (ie. sensor/actuator thresholds, base rate) to the Pacer process through message queues.




(a) Architecture Block Diagram




(b) Interface Module Block Diagram

Figure 4.10: Pacemaker Platform Instance

### 4.3.2 Architecture Implementation

In this section, we detail the architectural components that comprise our platform focusing on how the monitoring wrappers can enforce isolation. Our general platform principles are orthogonal to the specific chip fabrication process being used, albeit certain implementation details and the choice of available IPs necessarily depend on it. Our described prototype implementation is based on a Xilinx Virtex-5 FPGA [128].

Fig. 4.10(a) shows a block diagram for an example platform instance. For simplicity, the example is composed of only one hardware processor, one CPU and one external memory, but note that in our prototype implementation the number of CPUs, HW processors and memories is only limited by the available chip resources. Monitoring wrappers depend on the type of processor they control: *HW wrappers* and *CPU wrappers* are employed for hardware processors and CPU respectively. Each wrapper is comprised of two modules: an *interface module* and a *control module*. The interface module mediates access from the processor to the communication infrastructure and provides standardized communication services. Processes can only exchange information by transmitting data through their interface module.[5] This effectively prevents fault propagation since the interface module is able to reject any data transfer that would lead to a certificate violation. The control module implements the actual monitoring logic: it checks all data transfers performed by the interface module against the property specification included in the certificate, and enforces the process timing reservation. A more detailed description of the interface and control module is provided in Section 4.3.2. Similarly, each external memory is connected to the communication infrastructure by a memory controller; however, since memories are passive component, no monitoring is required. Finally, the platform includes three additional *global modules*: the *Communication Scheduler*, the *Frame Generator*, and the *Power Manager*, which are used to enforce communication timing reservation and power consumption. They are described in Section 4.3.2.

We can distinguish among four types of blocks in Fig. 4.10(a): 1) blocks that are generated by the designer. These include the hardware logic of each HW processor implementing the executed process and the code for each software process executed on a CPU. 2) Blocks that are automatically generated by our implementation tools.

[5]Note that since processors can implement external I/O, processes could potentially communicate through the environment. We assume that designers do not intentionally introduce such covert channels, and any environment dependency is solved through application of control theory.

These include the portion of the control module that implements the certificate and the Communication Scheduler. 3) Blocks that represent available IPs. These include: A) memories and their controllers; B) the communication infrastructure; C) CPU and associated Operating System (OS). 4) Blocks that are manually written by the platform provider, and depend both on the fabrication process and the supported IPs. These include the wrappers, Frame Generator and Power Manager. We implemented these blocks in a parameterized, mixed VHDL/Verilog Register-Transfer-Level (RTL) description.

The platform can support a variety of different IPs, albeit due to time constraints we only implemented wrapper support for some of them. However, specific constraints must be imposed on IPs to ensure required isolation and timing predictability. In what follows, we specify the required constraints for A) memories, B) communication infrastructure and C) CPU and OS. A) The access time to external memories should be predictable, or at least upper bounds must be easily computable. Our pacemaker prototype employs an external SRAM, which offers deterministic access time. Predictable DRAM controllers for real-time systems have been described in the literature [5]. B) Shared buses, segmented buses and Networks-on-Chips (NoCs) can all be employed as communication infrastructure, but the specific IP choice impacts the derivation of the communication schedule as described in [101], but not included here because it is unrelated to understanding the function BusMOP within an SoC context. For simplicity, in the rest of the chapter we consider a simple shared bus, deferring the analysis of more complex infrastructures to future work; our prototype employs the IBM PLB [73]. C) We do not allow caches on CPU. While caches can greatly improve average computation times for general purpose computing, their inherent unpredictability makes it very hard to obtain tight computation time bounds for processes and to schedule cache miss traffic. Instead, we provide each CPU with local instruction and data memory used as *scratchpads*. Scratchpads can be as fast as caches, but data transfers to/from external memory must be explicitly initiated by the running process instead of being handled by the cache controller. While this imposes additional responsibility to the designer, advantages in term of timing predictability and simplicity of implementation are well documented both in academia [87] and industry [38]. Furthermore, a large number of embedded CPU available either as soft or firm IPs for SoC design have configurable memory paths, and can therefore be easily connected to custom-designed scratchpads. Our platform can implement separate instruction and data scratchpads to support Harvard CPU architectures. In particu-

lar, our prototype supports the Xilinx Microblaze soft CPU running the Xilkernel OS. We shall assume that the code of all processes and of the OS executed on the CPU fits in the instruction scratchpad, which is typically possible thanks to the small footprint of embedded processes. The data scratchpad is controlled by the interface module: the processor can request it to transfer data from/to any external memory to/from the data scratchpad. The designer specifies the size of both the instruction and data scratchpad as part of the mapping process. Finally, if multiple processes are executed on the same processor, additional isolation requirements are imposed on the CPU and OS. In particular, the CPU must provide memory protection since scratchpads are shared. Furthermore, the OS must provide strong code isolation, especially regarding shared memory structures and libraries. A full description of OS-level isolation techniques is outside the scope of this thesis. As an example, the ARINC653 avionic standard [2] prescribes a set of requirements for the safe integration of mixed-criticality applications on a shared CPU, and ARINC653 certified OSs are available on the market.

An important note is relative to the criticality of the various platform blocks. Any fault in the communication infrastructure, external memories and controllers, global modules or in any monitoring wrapper can potentially compromise the whole system. As such, these components must be formally verified and/or certified to the highest degree of safety required by any application executed on the platform. While this can be expensive for the communication infrastructure and memories, we argue that if any form of communication and memory sharing is required in a mixed-criticality system, such requirement can not be avoided. The monitoring wrappers and global modules, in particular the control module and all schedulers, has been designed to be relatively easy to certify. A formal proof of the correctness of generated runtime monitors is available [33], albeit this does not remove the need for certification of the final implemented wrappers, since it is unfeasible to formally verify FPGA implementation tools like the placer and router.

### Global Modules

Three global modules, the Frame Generator, Communication Scheduler and Power Manager, are included in each platform instance to manage system-wide behavior. Each HW component in our platform can be clocked independently. In particular, each monitoring wrapper employs two distinct clocks: a system_clk that is used for the control module and interface module, and has the same frequency for all wrap-

pers; and a proc_clk that is used for the processor. In this way, different CPU and HW processors can run at different frequencies. As the integration scale of SoC becomes larger and clock frequency increases, it becomes impossible to synchronize all HW modules based on the same clock because signal propagation delay grows larger compared to the clock period. As such, large ASIC designs are moving towards a Globally-Asynchronous, Locally-Synchronous (GALS) paradigm where individual modules are based on synchronous logic but inter-module communication is asynchronous. Unfortunately, asynchronous communication is ill-suited to safety-critical systems: formally verifying asynchronous systems is order of magnitudes more complex when compared to synchronous design.

To solve this problem, we divide time into fixed-length intervals which we call *frames*. The Frame Generator periodically produces a frame_signal that is propagated to all wrappers; the frame period is significantly larger than both the physical clocks (in our prototype, we use a frame period of 1us, while the system_clk has a 8ns period) and signal propagation delays. All timing requirements (period, deadline and execution time) expressed in the AADL model are multiples of the frame period, and processors are synchronized based on frames. In particular, processes are periodically activated and communication is initiated on a frame boundary.

The Communication Scheduler uses the frame information to control data transmission on the communication infrastructure. For each process, a frame_grant signal is propagated to the corresponding monitoring wrapper: during each frame, the interface module is allowed to transmit on behalf of the process only if the provided frame_grant signal is set. Internally, the Communication Scheduler implements a scheduling table based on a fixed length hyperperiod: for each frame in the hyperperiod, the table determines which processes are allowed to transmit. In practice, since a same set of processes could be allowed to transmit for an interval of several frames in a row, the table encodes the length of every such interval. To ensure timing predictability, the table is built to enforce *contentionless communication*: two processes can be allowed to transmit in the same frame only if their data transmissions can be carried out in parallel. In particular, since our prototype employs a shared bus, we allow a single process to transmit in each frame. If for example the communication infrastructure is implemented as a segmented bus connected by bridges, processes using different bus segments could be allowed to transmit simultaneously. Note that apart from timing predictability, the contentionless principle can simplify infrastructure design: for example, wormhole routers with no packet buffers can be used in a NoC [55].

Finally, the Power Manager monitors power consumption. A clk_en signal is propagated to each monitoring wrapper. The proc_clk is periodically generated only while the clk_en is high; therefore, the Power Manager can completely stop a processor by simply lowering the corresponding clk_en signal. In our implemented prototype, the Power Manager periodically checks the input voltage to the system using the System Monitor functionality of the Virtex-5 FPGA; in a battery-operated system such as a pacemaker, this can be used to derive an estimate of remaining battery energy. If the energy becomes dangerously low, the Power Manager can shut down the non-life critical systems running on the CPU to save energy for the base pacing module. If additional measurement functionalities were available on the chip, the Power Manager could implement more refined control actions. For example, if power consumption could be measured for each processor, a misbehaving processor consuming an excessive amount of energy could be selectively turned off.

### Interface and Control Module

Fig. 4.10(b) shows a detailed block diagram of the interface module for a HW processor: it is composed of three main submodules, the *Processor Interface*, the *DMA Engine*, and the *Communication Interface*. The Processor Interface is directly connected to the processor and exports the communication services; it uses the same proc_clk as the processor. The DMA Engine uses the system_clk and implements most of the interface module logic. Finally, the Communication Interface connects to the communication infrastructure and shares its physical clock: it converts data transfer commands issued by the DMA Engine into read/write transactions on the communication infrastructure. This division of concerns simplifies development and certification: a new Communication Interface must be implemented for each communication infrastructure IP, but the DMA Engine is fully reusable. The Communication Interface receives the frame_grant as input, and it is allowed to read/write only when the signal is high. In practice, to account for the effect of propagation delay, after frame_grant goes high the Communication Interface must wait for a guard time equal to twice the maximum signal propagation delay before it can start to transmit.

Since all three submodules lie in different clock regions, dual-port memory elements are used to connect them. The *data scratchpad* can be simultaneously accessed by both the Processor Interface and DMA Engine. A variable number of *FIFO queues* are used to hold incoming data to message queues; the processor can

read from the FIFO queues at any time. Service request by the processor are sent to the *request queue*. There are two types of requests: 1) sending a message to the message queue of another processor; 2) performing either a read (from external memory to data scratchpad) or write (viceversa) DMA operation. A message transfer request contains an ID for the destination queue and the data to be sent, while a DMA request contains the length of the transfer and the base addresses in both external memory and the scratchpad. All transfers are multiple of a 32-bits word. The *completion queue* is used to signal the end of a DMA operation back to the processor. Finally, the *transfer queue* is used to connect the DMA Engine to the Communication Interface.

The DMA Engine processes incoming service requests in order based on the request type. First of all, for every message transfer it translates the queue ID into an address. All external memories and message queues are automatically assigned unique system-wide addresses by our tools. The address is used by the communication infrastructure and Communication Interface to determine the destination of a data transfer. Second, for message transfers and DMA write, the DMA Engine simultaneously moves data from the request queue or the data scratchpad to the transfer queue, and sends the data to the *Run-Time Monitoring Engine* in the control module. The engine is responsible for checking each transferred data word against the functional properties included in the certificate, and it issues either a reject or accept command in 4 clock cycles, and it uses monitors generated by BusMOP. If the transfer is rejected, the transfer queue is flushed. Otherwise, the DMA Engine commands the Communication Interface to start the data transfer. The DMA Engine then immediately proceeds to service the next request; in particular, to avoid stalling the Communication Interface, it can write to the transfer queue while a previously accepted data transfer is being carried out. Incoming data from DMA read operations is forwarded to the Run-Time Monitoring Engine but never rejected, hence a reception queue is not required; as explained in Section 4.4, this is allowed because read operations can not cause side-effects in the system. Similarly, incoming messages are monitored but immediately injected into the corresponding message queue.

The control module for a HW processor is composed of two main submodules. The aforementioned runtime monitoring engine is described in more details in Section 4.4. The process scheduler receives the frame_signal as input and periodically activates the HW process through a process_start wire. The process is responsible for signaling the end of its periodic execution through a process_stop

signal. Failure to do so denotes a critical error; as a consequence, the process is stopped by halting its proc_clk clock.

The interface and control module for a CPU processor have some added complexity, mainly because multiple processes can be executed on the same CPU. Message queues, request and completion queues, transfer queues and Run-Time Monitoring Engines are duplicated for each process; our wrapper VHDL description is parametric in the number of processes and our Microblaze implementation supports up to ten. Note that we expect few processes to be mapped on each CPU (typically one per executed application), since each process can have multiple threads. One frame_grant signal for each executed process is provided to both the DMA Engine and Communication Interface; they each service the process for which the frame_grant is high. The Processor Interface is customized based on the CPU IP; in our implementation for the Microblaze all services are exported through memory-addressable registers. Finally, the process scheduler is replaced with a CPU scheduler. It implements a scheduling table for processes which is similar to the table in the Communication Scheduler. Whenever a different process must be executed, the CPU scheduler interrupts the CPU and communicates the ID of the process. In this way we make sure that computation is synchronized at the frame boundary, while requiring only minimal kernel modification. Note that while we do not support it in our Xilkernel implementation, the OS can implement a two-level scheduler (as required for example by ARINC653), where the time assigned to each process by the CPU Scheduler is distributed to its threads according to a low-level, process-specific software scheduler.

**Implementation Results**

We have used our developed tools to automatically generate wrappers based on the AADL pacemaker specification described in Section 4.3.1. The platform was instantiated and synthesized using Xilinx Embedded Development Kit 10.1.3 on a Virtex-5 VLX50T FPGA. We created both correct and faulty versions of low-criticality processes (in particular, the programmer process) to test the capability of the monitoring wrappers to reject faults. The design behaved according to specification, albeit more stringent testing would be required for certification.

Since propagation delay is less of a concern in current FPGA chips, a single 125Mhz system_clk is shared among all wrappers. The VLX50T FPGA can support up to 11 additional clocks for processors and memories; in our pacemaker

implementation, the CPU and all HW processors use separate proc_clk clocks also running at 125Mhz. The communication infrastructure can transfer up to 4 bytes per cycle, for a total bandwidth of 500MB/s; however, due to guard time, a maximum of 480 bytes can be transferred every 1us frame, resulting in an effective bandwidth of 480MB/s. Finally, we measured an OS footprint of 7648 bytes in instruction and 15234 bytes in data scratchpad.

## 4.4 Functional Isolation

Our approach to provide functional isolation relies on runtime monitoring. As described in Section 4.3, for each process the designer specifies a set of formal properties that describe its allowed communication behavior. At runtime, the monitoring engine in the control module checks that all properties are satisfied. If a violation is detected, then the control module can take a suitable *recovery action* to keep the system in a safe state; in particular, any faulty data transfer can be rejected before it is propagated on the communication infrastructure.

Our toolset is based on Monitoring-Oriented Programming (MOP) (see Chapter 2). MOP is a highly extensible and configurable Runtime Verification framework. The user is allowed to extend MOP with his/her own logics via *logic plugins* which encapsulate the monitor synthesis algorithms. Currently, we support both Past-Time Linear Temporal Logic (PTLTL) and Extended Regular Expressions (ERE). Property specification consists of event definitions and logical formulae or patterns. The formula or pattern designates which "traces" (observed series of events) are valid or invalid. Recovery can be initiated either when a validation or violation of the trace is detected. For EREs, valid traces are those which are strings in the language represented by the ERE, with events treated as the letters in the alphabet of the language. Neutral traces are prefixes of strings in the language, while violations are invalid strings. For PTLTL formulae, valid traces are any traces for which the formula evaluates to true, invalid traces are those for which the formula evaluates to false; there are no neutral traces. For more information on regular languages and temporal logic see Chapter 5.

**Illustrative Example.** An example of property specification for the Event Tagger process can be seen in Fig. 4.11. Recall from Section 4.3 that two data objects are placed in external memory: the signalBuffer and the logBuffer. The property, called SAFEMEMORYACCESS, makes sure that the Event Tagger can

```
soc SafeMemoryAccess{
  signal numFailures : STD_LOGIC_VECTOR(11 downto 0) := X"000";

  event logBufferWrite : write in logBuffer
  event signalBufferWrite : write in signalBuffer

  ere: (logBufferWrite)*

  @violation {
    reject_reg <= '1';
    if(numFailues >= 1000) then
      reset_reg <= '1';
      numFailures <= (others => '0');
    end if;
  }
}
```

Figure 4.11: Example Property: SAFEMEMORYACCESS

not erroneously overwrite the data written by the critical Pacer module in the signalBuffer. Each property is comprised of a formula, an EventList, a declaration section and validation/violation handlers. In this example, the declaration section simply introduces one monitor-local register that keeps track of the number of erroneously attempted writes. The two events, signalBufferWrite and logBufferWrite, capture writes to the signalBuffer and logBuffer respectively. The ERE pattern matches zero or more occurrences of the logBufferWrite event; hence, any write to signalBuffer will cause a violation. Recovery is specified in the violation handler: the data transfer is rejected. Furthermore, after 1000 erroneous writes, presumably indicating the presence of a bug rather than a transient fault, the process is reset.

As described in Section 4.2, BusMOP generates a VHDL module for each property. We have developed new logic that connects all generated property modules together in the Monitoring Engine. Compared to PCI Bus monitoring, this implementation has two important novelties. First of all, BusMOP for PCI bus monitoring could only recover through corrective actions (in particular, overwriting memory locations): this is because data transfers could not be prevented from being propagated on the bus. In our SoC platform, faulty data transfers can be rejected in the interface module. Second, since properties are specified in the AADL model, events can be expressed using the symbolic names of message queues and data objects; our tool then automatically translates from names to corresponding addresses. In BusMOP, the designer had to manually specify all memory addresses, which is error prone.

**Event Specification.** Again, a formal syntax of event specifications can be see in Chapter 2. The only difference is that within the SoC framework, names can be

used for memories rather than numerical addressed.

All events are read/writes from/to either a message queue or a data element in external memory, identified by its name in the AADL model. Normally, an event is triggered whenever any word of a data element is changed. However, the designer can specify a numerical address inside the data element, in which case the event is triggered only when that specific memory location is read/written. This can useful to access individual components of a complex data structure. In this case, the designer can also specify a desired value range. Ranges can consist of a single arithmetic expression, or a pair of comma separated arithmetic expressions denoting the minimum and maximum values that can trigger the event. Since all transfers in the system are multiple of the word size, values are always assumed to be 32-bits.

**Recovery.** Recovery actions in the handlers are specified as a list of concurrent VHDL statements. Several recovery actions are possible in the handler: 1) the current data transfer can be rejected in the interface module; 2) the monitored process can be stopped. In a HW processor, this is achieved by stopping the proc_clk. In a CPU, the output of the CPU Scheduler is altered so that the processor is never executed; 3) the process can be reset. The reset functionality depends on the design of the processor. HW processors implemented on FPGA use synchronous logic and are provided with a reset signal from the control module. In our Microblaze implementation, the control module can interrupt the CPU and signal the reset action. Xilkernel can then kill the process and restart it. 4) The DMA Engine can be instructed to carry out a send to a message queue or a write to external memory. Monitor requests take precedence over all other data transfer requests, but they can still be carried out only during a frame assigned to the process by the Communication Scheduler. The set of registers described below is used to specify the recovery action in VHDL; as with events, symbolic names can be used in place of addresses.

| | Data Transfer Interface |
|---|---|
| address_reg | address |
| value_reg | value send/written |
| execute_reg | start transfer |
| reject_reg | reject transfer |
| stop_reg | stop process |
| reset_reg | reset process |

Note that only outgoing data transfers can be rejected; incoming data is always

accepted. By checking all outgoing transfers, we always make sure that any data propagated on the communication infrastructure conforms to specified certificates. Hence, incoming data must also conform to system specification. Furthermore, since timing isolation is always enforced, DMA read operations can not steal resources or change the state of other processes.

**Programmer Process.** We now describe a more detailed example to show how monitors can be directly exploited to enforce correct communication between lower and higher criticality applications. As described in Section 4.3, the Programmer process is used to update execution parameters of both the Pacer and Rate Adapter based on received RF commands. This is potentially dangerous, because both the Pacer and Rate Adapter are more critical than the Programmer and RF process. To solve the problem, we can introduce a commit protocol. The Programmer sends new parameters followed by a check command to the Rate Adapter through the *RateAdapter.parameters* message queue. The Rate Adapter validates the received parameters and sends back either a success or a failure answer to the *Programmer.response* message queue. The Programmer then repeats the same steps for the Pacer using its *Pacer.parameters* queue. If the Programmer receives success answers from both processes, it then sends commit messages to the Rate Adapter and Pacer process, causing them to load the received parameters. Unfortunately, since the Programmer is a complex, non safety-critical process, it could fail after sending a commit command to just one of the two processes. While this would not compromise the life-critical functionality (the Pacer control algorithm rejects any unsafe control points [11]), it can nevertheless disrupt the Rate Adapter causing significant discomfort to the patient.

```
checkPacer   : write at Pacer.parameters       value in X"40000000"
checkRate    : write at RateAdapter.parameters value in X"40000000"
successPacer : read  at Programmer.response    value in X"80000000"
successRate  : read  at Programmer.response    value in X"20000000"
failurePacer : read  at Programmer.response    value in X"40000000"
failureRate  : read  at Programmer.response    value in X"10000000"
commitPacer  : write at Pacer.parameters       value in X"80000000"
commitRate   : write at RateAdapter.parameters value in X"80000000"
```

Figure 4.12: Event List

The solution that we adopt is to send the commit command directly from the monitor; in this way, we isolate the critical functionality of the Programmer module inside the certified wrapper. A set of Programmer events are specified in Fig. 4.12, consisting of check and commit commands and success and failure answers for

```
...
1. ptltl : successPacer and <*>successRate and
2.   (*)( not (not(successRate) S successPacer) ) and
3.   (*)( not (not(successRate) S failureRate) )

@validation {
  address_reg <= Pacer.parameters;
  value_reg   <= X"800000";
  execute_reg <= '1';
}
...
```

Figure 4.13: SENDPACERCOMMIT Specification

```
...
1. ptltl : commitRate or commitPacer or (
2.   (checkRate or checkPacer) and
3.   (*)(
4.     (not(successRate or failureRate) S checkRate)  or
5.     (not(successPacer or failurePacer) S checkPacer)
6.   )
7. )
@validation { reject_reg <= '1';}
...
```

Figure 4.14: CHECKPROGRAMMERCOMMANDS Specification

both the Pacer and the Rate Adapter. The *SendPacerCommit* property in Fig. 4.13 is used to send the commit command to the Pacer on validation (an equivalent property with different handler can be used to send the commit to the Rate Adapter). In the PTLTL formula, <*>, (*) and S are temporal operators denoting *eventually in the past*, *previously* and *since*. Line 1 of the formula specifies that a success is received from the Pacer in the present and a success from the Rate Adapter has been received in the past. Line 2 implies that at least one successRate event has been received since the previous successPacer (if any), and Line 3 implies that at least one successRate has been received since the previous failureRate (if any); this makes sure that a valid parameter set has been passed to the Rate Adapter since the last commit operation. Finally, property *CheckProgrammerCommands* in Fig. 4.14 is used to reject any erroneous Programmer command. Line 1 is used to reject commit commands from the Programmer, since only the monitor should send them. Lines 2-7 make sure that the Programmer can not send a check command if it has not received an answer (either a success or a failure) to its previous check commands to both the Pacer and Rate Adapter: otherwise, the Programmer could send a check command immediately before a commit is sent by the monitor, causing a wrong set of parameters to be loaded.

## 4.5 Chapter Related Work

There are two main Runtime Verification approaches: 1) offline, where a log, or trace is kept, which can then be used for purposes of debugging; and 2) online, where a property is checked while the program is running. As BusMOP is an online technique, we will only describe online approaches to Runtime Verification. This is a short related work, a summary with more related work on monitoring in general can be found in Chapter 1.

MaC [82], Tracematches [10], Eagle [16], and all the various other runtime monitoring systems of which we are aware use specific verification languages which cannot be changed, while BusMOP, as an extension of MOP [93], will eventually support all the finite logics supported in JavaMOP. Temporal Rover [44] is a commercial Runtime Verification tool which uses future time metric temporal logic. It provides inline specification of monitors, where the monitors are written straight in the source file. Inline specification does not make sense for BusMOP, as there is no program being monitored per se. Program Query Language (PQL) [89], is an approach somewhat similar to MOP, although it also only allows one specification language. PQL can support the full generality of context free languages. Tracematches is very similar to JavaMOP. The biggest difference is that its choice of regular expressions for logical formalism is hardwired. It is an extension of the AB [9] AspectJ compiler. All of the above approaches are designed to monitor specific programs, and are implemented in software. This has the effect of both adding runtime overhead, and performing a function different from that of BusMOP, which monitors COTS peripherals.

The PSL to Verilog compiler, P2 [88], is the sole attempt to perform formal Runtime Verification in hardware, of which we are aware. P2V is similar to BusMOP in that monitors are implemented in hardware rather than software, and that both approaches thus have no runtime overhead on the CPU. P2V, however, is more like the above approaches in that it is designed for monitoring actual programs rather than peripheral devices. Also it requires a dynamically extensible soft-core processor implemented on an FPGA, while our approach can potentially be applied to any COTS communication architecture. Further, P2V uses hardwired logic while BusMOP allows different formalisms.

The concept of PBD is well established and several SoC platforms are commercially available [40, 96]; however, they are not targeted at safety-critical systems. Similarly, a variety of PBD and model-driven design methodologies have been

proposed (see [107] for an overview), several of which (for example, see [12, 78]) support formal verification of modeled application behavior. The main difference is that our methodology and platform does not simply allow to verify the correctness of a high-level model: it enforces application isolation at runtime, as long as the designer can correctly specify system isolation requirements in the model.

## 4.6 Chapter Conclusion

The efficacy of the BusMOP system has been demonstrated in a hardware and hardware/software domain through two usages: monitoring bus traffic for PCI COTS components, and providing functional isolation in System-on-Chip (SoC) design. This shows that Runtime Verification is effective outside of its originally intended software environment.

COTS peripherals are increasingly being adopted in the embedded market for performance reasons. However, COTS components introduce challenges in the development of critical systems, as they are unpredictable and often complete hardware specification is not publicly available. In this chapter, we have proposed runtime monitoring of bus activities as a way to cope with such unpredictability. A monitoring device can be plugged on a PCI bus segment and check that all communication between peripherals and the rest of the system behaves according to specifications. Monitoring logic is automatically generated by the BusMOP framework and synthesized on FPGA, resulting in zero CPU runtime overhead. Finally, we showed the applicability of our monitoring infrastructure and recovery mechanisms on a real test case.

Implementing safety-critical embedded systems like medical devices as SoC is promising, but unfortunately there is a lack of suitable design methodologies. Integrating mixed-criticality systems is especially challenging because low-criticality IPs are too expensive to be verified and/or certified, but they must be prevented from interfering with high-criticality applications. To solve this problem, a new design methodology and architectural platform were introduced. The key idea is that our platform supports behavioral enforcement through BusMOP: low-criticality processes are guaranteed to behave according to a published certificate. This does not remove the need to certify high-criticality components and verify correct system-level behavior, but it enables the design to do so without worrying about unpredictable faults in low-criticality components.

We plan to extend this work in two directions. From a system point of view, we plan to develop a interposing PCI/PCI-X/PCI-E monitoring device capable of executing preventive recovery actions as described in Section 4.2.2. While the SoC case study shows that preventative actions are feasible, we believe a PCI/PCI-X/PCI-E monitoring device capable of preventative action would be highly desirable. From a formal specification point of view, we plan to extend BusMOP to support all MOP logic plugins through the current MOP FSM plugin.

# Chapter 5

# Finite Logics

## 5.1 Chapter Introduction

In this chapter, we discuss the finite-state logic plugins available in MOP. Each logical formalism provided by the MOP framework is implemented as a program called a *logic plugin*, as mentioned in Chapter 2. The individual logic plugins are controlled by the logic repository as can be seen in Fig. 2.1.[1]

For each plugin we first discuss the syntax of specifications using the formalism. Each plugin syntax instantiates the generic ⟨*Logic Name*⟩, ⟨*Logic Syntax*⟩, ⟨*Logic State*⟩ non-terminals described in Section 2.5.1 (see also Fig. 2.3). Internally, aside from the specified syntax, each plugin is also given the set of events used in the property by the instance client (either JavaMOP or BusMOP). The instance client simply drops the instance-specific event definition and actions, which are irrelevant to the plugin, and sends only the event names. For simplicity, in what follows, the event declarations are dropped from the logic plugin's syntax. To clarify the syntax we show an example property in each logic, which also does not include event descriptions. Because only the property is shown, without any event definitions, the parameters, which are part of event definitions in JavaMOP, are absent. One should be aware, if one wishes to use the logic repository described in Chapter 2 without using one of the two pre-defined instance clients, that one should also provide the event names with the property.[2]

We then discuss some issues specific to the particular plugin as well as how monitor pseudocode suitable for conversion to Java and HDL is generated. The

[1] The Finite State Machine (FSM) plugin is work with Grigore Roşu that appeared in [93] the extended-regular expressions plugin is a reimplementation of an older algorithm [111], past time linear temporal logic (PTLTL) is an extension of an older algorithm, and linear temporal logic with past builds upon the work of [53] and [42] to provide the first monitoring algorithm for LTL with both past and future operators.

[2] Additionally, one must use the logic repository XML syntax, which distinguishes the events from the property.

⟨*FSM Name*⟩ ::= "*fsm*"
⟨*FSM Syntax*⟩ ::= {⟨*Item*⟩}{⟨*Alias*⟩}
⟨*Item*⟩ ::= ⟨*State Name*⟩ "[" { ⟨*Transition*⟩ [","]} "]"
⟨*Transition*⟩ ::= ⟨*Event Name*⟩ "–>" ⟨*State Name*⟩
| "*default*" ⟨*State Name*⟩
⟨*Alias*⟩ ::= "*alias*" ⟨*Group Name*⟩ "="
{ ⟨*State Name*⟩ "," } ⟨*State Name*⟩
⟨*FSM State*⟩ ::= ⟨*Group Name*⟩ | ⟨*State Name*⟩ | "*fail*"

Figure 5.1: FSM Syntax

pseudocode generated from each example property is shown to make the explanation of monitor pseudocode generation more concrete.

For every plugin we also describe how enable sets are generated. Recall that enable sets distill information about which prior events must be seen for a given event to create a new monitor instance (see Defs. 10, 11, 12 and the surrounding text in Sections 3.3 and 3.3.2, all from Chapter 3, for a complete explanation of enable sets).

### 5.1.1 Chapter Contributions

This chapter presents algorithms for minimizing multicategory finite state machines, as well as enable set calculations for finite state machines. It contains a description of a reimplementation of the translation from extended regular expressions to finite state machines, a parallel algorithm for efficiently monitoring past time linear temporal logic in hardware, and the first implementation of a monitoring algorithm for linear temporal logic with both future and past operators.

## 5.2 Finite State Machines

The finite state machine (FSM) plugin is one of the most important plugins for MOP. Not only is it a useful logical formalism in itself, but it is used as a backend for all logics reducible to finite automata. Currently, all logic plugins except for the context-free grammar (CFG; Chapter 6), string rewriting systems (SRS; Chapter 7), and past time linear temporal logic with calls and returns (PTCaRet; see [105] and [93]) generate FSM output. This allows for a strong separation of concerns. For instance, minimization need occur only once, and it allows us to use one enable set generation algorithm for all of these plugins. Additionally, each

```
start [
  default start
  next -> unsafe
  hasnext -> safe
]
safe [
  next -> start
  hasnext -> safe
  dummy -> safe
]
unsafe [
  next -> unsafe
  hasnext -> safe
]
alias all_states = start, safe, unsafe
alias safe_states = start, safe
```

Figure 5.2: FSM Example

instance of MOP need only know how to translate the pseudocode for FSMs, CFGs, SRSs, and PTCaRet. Some MOP instances, such as BusMOP, may even opt to support only finite state monitors, in which case they only need to provide support for translating FSM pseudocode.[3]

Fig. 5.1 shows the syntax for FSM properties. An FSM property is a series of ⟨*Item*⟩s followed by ⟨*Alias*⟩s. An ⟨*Item*⟩ is essentially a state in the finite state machine, and the different transitions to take on a given input (⟨*Transition*⟩). The ⟨*Alias*⟩ allows for giving a name to a *set* of states. This is invaluable, because the ⟨*FSM State*⟩ non-terminal, which defines what categories may trigger handlers, both ⟨*Group Name*⟩s, which are the names associated to sets of states in ⟨*Alias*⟩s, and ⟨*State Name*⟩s may be associated with handlers. This allows one to write a property that triggers actions when any state in a given ⟨*Alias*⟩ is entered.

Fig. 5.2 shows an example FSM property. In this example, three events: next, hasnext and dummy, and three states: start, safe and unsafe are defined. Two state aliases are declared: all_states represents all the states in the state machine and safe_states includes the start state and the safe state. The fail category is reported whenever an event occurs that is not specified for the current state. For example, the state machine will go into fail when the dummy event is seen in the unsafe state. The default transition in the start state covers any event not specified in the

[3]Though note that in the case of BusMOP, finite state machines are not used for PTLTL in favor of using parallel assignments (see Section 5.3.1).

transition. Because of this, any state with a default transition cannot lead to a fail category for any input. As mentioned in Fig. 5.1, handlers may be associated with any state (e.g., start) or group name (e.g., all_states).

In the interest of keeping runtime monitoring as efficient as possible, we wish to use minimized finite state machines for monitors. Because of the ability to trigger handlers from ⟨*State Name*⟩s and ⟨*Group Name*⟩s, MOP FSM properties are *multicategory* finite state machines (finite state machines that recognize more than one language; essentially equivalent to Moore machines). This requires a small change to the normal Hopcroft FSM minimization algorithm [65].

The Hopcroft algorithm works by assuming the largest possible equivalence class of states, and then partitioning the equivalence classes into smaller classes if necessary. The way the algorithm determines that it is necessary to split is by considering two equivalence classes $C_1$ and $C_2$ and an input, $e$. For each state $s$ in $C_1$, if s goes to a state in $C_2$ on $e$ then it goes into class $C_{11}$, otherwise it goes to class $C_{12}$. Classes are continuously split by other classes until a fixed point is reached. When a fixed point is reached, each equivalence class becomes a state in the final machine.

The way our algorithm differs is in the initial partition. The normal algorithm partitions the states into two classes, those states that are final states and those which are not. We, however, have multiple categories. The particularly interesting feature, is that categories may overlap on states. If two categories $C_1$ and $C_2$ overlap, they must have three equivalence classes: those states in $C_1 - C_2$, those in $C_2 - C_1$, and those in $C_1 \cap C_2$. The naive algorithm would be to compute the intersections between all the categories, but that is quadratic in nature. A better algorithm, which we use, is to find the set of categories each state belongs to. This takes time linear in the number of states. Those states that have the same set of categories are placed in the same initial equivalence class.

The monitor pseudocode for an FSM property appears the same as the input code, except that it will be minimized, whereas the input code may not be minimal. Because of this, we omit the output of the example in Fig. 5.2. Each MOP instance that wants to support the finite state machine-based logics must convert the FSM pseudocode into executable code.

The algorithm in Fig. 5.3 computes the property enable sets for a finite state machine [31]. We use this algorithm to compute the enable sets for any logic that is reducible to a finite state machine, including ERE, LTL, and PTLTL. The algorithm assumes a finite state machine, defined as $FSM = (\mathcal{E}, S, s_0 \in S, \delta : S \times \mathcal{E} \rightarrow S)$.

```
Algorithm 𝓔𝓝_fsm(FSM = (𝓔, S, s_0, δ))
Globals: mapping 𝒱_μ : S → 𝒫_f(𝒫_f(𝓔))
         mapping enable^𝓔_𝒢 : 𝓔 → 𝒫_f(𝒫_f(𝓔))
Initialization: 𝒱_μ(s) ← ∅ for any s ∈ s
                enable^𝓔_𝒢(e) ← ∅ for any e ∈ 𝓔
function main()
 1 compute_enables(s_0, ∅)
function compute_enables(s, μ)
 1 for all defined δ(s, e) do
 2 ⋮ enable^𝓔_𝒢(e) ← enable^𝓔_𝒢(e) ∪ {μ}
 3 ⋮ let μ' ← μ ∪ {e}
 4 ⋮ if μ' ∉ 𝒱_μ(s)
 5 ⋮ ⋮ 𝒱_μ(s) ← 𝒱_μ(s) ∪ {μ'}
 6 ⋮ ⋮ compute_enables(δ(s, e), μ')
 7 ⋮ endif
 8 endfor
```

Figure 5.3: FSM $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ Computation Algorithm [31].

$\mathcal{E}$ is the alphabet, traditionally listed as $\Sigma$ but changed for consistency, because the alphabets of our FSMs are event sets. $s_0$ is the start state, corresponding to 1 in the definition of a monitor. $\delta$ is the transition partial function, taking a state and an event and potentially mapping to a next state for the machine. Note that we can extend $\delta$ to be consistent with $\sigma$ in Def. 7 by simply completing the function by adding an undefined state, *undef*, and making all non-existent transitions point to *undef*. This is how FSMs are handled in the JavaMOP instance. We assume that all states not reachable from the initial state and not coreachable from the states of interest (states of interest being those states $s$ such that $\gamma(s) \in \mathcal{G}$, where $\mathcal{G}$ is the goal category; see Def. 11) are pruned from the FSM before running the algorithm, leaving the transitions that pointed to them undefined. $\mathcal{V}_\mu$ is a mapping from states to sets of events; it is used to check for algorithm termination. $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ is the output property enable set, which is converted into a parameter enable set by the language instance client, discussed in Chapter 2.

Function compute_enables is first called from main with $\mu = \emptyset$ and the initial state $s_0$. If we think of the FSM as a graph, $\mu$ represents the set of edges we have seen at least once in a given traversal path. For each defined $\delta(s, e)$ (line 1), we add the current $\mu$ to the $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e)$ (line 2) because this means we have seen a viable

⟨*ERE Name*⟩ ::= "*ere*"
⟨*ERE Syntax*⟩ ::= "*empty*" | "*epsilon*"
| ⟨*Event Name*⟩
| ⟨*ERE Syntax*⟩ "*"
| ⟨*ERE Syntax*⟩ "+"
| "~" ⟨*ERE Syntax*⟩
| ⟨*ERE Syntax*⟩ "|" ⟨*ERE Syntax*⟩
| ⟨*ERE Syntax*⟩ "&" ⟨*ERE Syntax*⟩
| ⟨*ERE Syntax*⟩ ⟨*ERE Syntax*⟩
| "(" ⟨*ERE Syntax*⟩ ")"
⟨*ERE State*⟩ ::= "*match*" | "*fail* | "*?*"

Figure 5.4: ERE Syntax

prefix set (as all non-viable paths in the machine have been pruned). This follows from the definition of $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$. Line 3 begins the recursive step of the algorithm. We let $\mu' = \mu \cup \{e\}$, because we have traversed another edge, and that edge is labeled as $e$. The map $\mathcal{V}_\mu$ tells us which $\mu$ have been seen in previous recursive steps, in a given state. If a $\mu$ has been seen before, in a state, taking a recursive step can add no new information. Because of this, line 4 ensures that we only call the recursive step on line 6, if new information can be added. Line 5 keeps $\mathcal{V}$ consistent. Thus the algorithm terminates only when every viable $\mu$ has been seen in every reachable state, effectively computing a fixed point. Thus, the algorithm is bounded by the number of one cycle paths through the graph (and is faster in practice, because most paths will have repeated events).

## 5.3 Extended Regular Expressions

Regular expressions can be easily understood by the average software engineer or programmer, as shown by the immense interest in and the success of scripting languages like Perl, based essentially on regular expression pattern matching. We believe that regular expressions provide an elegant and powerful specification language also for monitoring requirements, because an execution trace of a program is in fact a string of states. Extended regular expressions (EREs) add complementation to regular expressions, which brings additional benefits by allowing one to specify patterns that must not occur during an execution. Complementation gives one the power to express patterns on strings non-elementarily more compactly. Also, one important observation about the use of ERE in the context of Runtime

$$
\begin{array}{rcl}
(R_1|R_2)\{e\} & = & R_1\{e\}|R_2\{e\} \\
(R_1R_2)\{e\} & = & (R_1\{e\})\ R_2 \\
 & | & \text{if}(epsilon \text{ in } R_1) \text{ then } R_2\{e\} \text{ else } empty \\
R*\{e\} & = & (R\{e\})R* \\
\tilde{}R\{e\} & = & \tilde{}(R\{e\}) \\
e_1\{e_2\} & = & \text{if}(e_1 = e_2) \text{ then } epsilon \text{ else } empty \\
epsilon\{e\} & = & empty \\
empty\{e\} & = & empty
\end{array}
$$

Figure 5.5: ERE Derivative Equations

Verification is that ERE patterns are often used to describe buggy patterns instead of desired properties.

Fig. 5.4 shows the syntax for ERE properties. The operators are standard for regular expressions, except that "~" is the language complement of an ERE, and "&" is language intersection. While "*epsilon*" is the empty string, as is normal, "*empty*" refers to the empty language.

Here is an example ERE property for the UnsafeMapIterator property previously shown in Fig. 3.3:

```
create_coll update_map* create_iter
use_iter* update_map+ use_iter
```

Recall that in this property the sequence of actions of importance is the creation of an Iterator from a Collection that was created from a Map, which is updated between the creation of the Iterator and its use.

FSMs are generated from EREs using coinductive techniques [111]. Briefly, in our approach we use the concept of derivatives of a regular expression, which is based on the idea of event consumption, in the sense that an extended regular expression $R$ and an event $e$ produce another extended regular expression, denoted $R\{e\}$, with the property that for any trace $w$, trace $e\ w$ is in $\mathcal{L}(R)$ (i.e., the language denoted by $R$) iff $w$ is in $\mathcal{L}(R\{e\})$. Fig. 5.5 defines this derivative semantics recursively on the structure of regular expressions; the operators without equations can be defined in terms of operators that do have equations specified. In the equations "|" refers to the ERE operator "|". The generated FSM is not minimal; the minimization algorithm of the FSM plugin (Section 5.2) is used to make it minimal. A deterministic automaton is produced, saving memory and time (in contrast to the more conventional Thompson approach [120], which operates by first producing a non-deterministic automaton, and then using a determinization

```
s0[
  create_coll -> s1
]
s2[
  use_iter -> s4
  update_map -> s3
]
s1[
  createIter -> s2
  update_map -> s1
]
s4[
  use_iter -> s4
  update_map -> s3
]
s3[
  use_iter -> s5
  update_map -> s3
]
s5[
]
alias match = s4, s5
```

Figure 5.6: ERE Example Output

⟨*PTLTL Name*⟩ ::= "*ptltl*"
⟨*PTLTL Syntax*⟩ ::= "*true*" | "*false*"
| ⟨*Event Name*⟩
| "*not*" ⟨*PTLTL Syntax*⟩
| ⟨*PTLTL Syntax*⟩"*and*" ⟨*PTLTL Syntax*⟩
| ⟨*PTLTL Syntax*⟩"*or*" ⟨*PTLTL Syntax*⟩
| ⟨*PTLTL Syntax*⟩"*xor*" ⟨*PTLTL Syntax*⟩
| ⟨*PTLTL Syntax*⟩"=>" ⟨*PTLTL Syntax*⟩
| "[*]" ⟨*PTLTL Syntax*⟩
| "<*>" ⟨*PTLTL Syntax*⟩
| "(*)" ⟨*PTLTL Syntax*⟩
| ⟨*PTLTL Syntax*⟩ "S" ⟨*PTLTL Syntax*⟩
⟨*PTLTL State*⟩ ::= "*validation*" | "*violation*"

Figure 5.7: PTLTL Syntax

algorithm to produce a deterministic automaton).

Fig. 5.6 shows the output FSM for the UnsafeMapIterator example ERE above. Note that the alias match is assigned so that ERE properties properly allow match as a verdict category. This was the initial motivation for aliases in the FSM plugin. Also note that the states do not have fully specified input, so fail is a possible output category.

As mentioned in Section 5.2, the property enable sets for EREs are computed by using the algorithm in Fig. 5.3 after the ERE has been converted to an FSM.

### 5.3.1 Past Time Linear Temporal Logic

Past time linear temporal logic is similar to standard linear temporal logic [102], except that all operators refer to the past. Some safety properties are more easily expressed in terms of the past than the future, for example the property that a user authentication be required before accessing some resource is most naturally expressed as "access => <*> authenticate", e.g., that an access requires an authentication at some point in the past. Monitors generated from PTLTL formulae also have the quality of validating or violating on every event because the past is already known. This contrasts with LTL monitors (see Section 5.4), which can also be in an intermediate, ?, state. Additionally, once an LTL monitor validates or violates, it is always violated or validated, whereas PTLTL is allowed to change on each event. PTLTL does not guarantee that its formula holds for all program executions, as one may expect from model checking, because a monitor never sees all possible paths through a program.

| Formula | Assignment | Initial $b[n]$ |
|---|---|---|
| $F_1\ S\ F_2$ | $b[n] \leftarrow B(F_1)$ and $b[n]$ or $B(F_2)$ | false |
| [*] $F$ | $b[n] \leftarrow B(F)$ and $b[n]$ | true |
| $<*>\ F$ | $b[n] \leftarrow B(F)$ or $b[n]$ | false |
| (*) $F$ | $b[n] \leftarrow B(F)$ | false |
| Formula | Expression | |
| $B(F)$ | $F$ if $F$ is a simple boolean formula, otherwise the $b[n]$ storing the value of $F$ | |

Figure 5.8: PTLTL Assignment Equations

Fig. 5.7 shows the syntax for our PTLTL plugin. The operators "*not*", "*and*", "*or*" , and "=>" (implication) are the standard boolean connectives. The operator "[*]" stands for "always in the past", meaning the formula following it must hold at all times in the past, while "$<*>$" stands for "eventually in the past", meaning that the formula following it must either currently hold or it must have held somewhere previously in the trace. The operator "(*)" means "previously": the formula following it, say $F$, must hold in the previous time step; in terms of MOP, this means that $F$ must have held when the previous event occurred. "S" means "since"; "$F_1$ S $F_2$" means "either $F_2$ must hold now, or $F_2$ must have held in the past and $F_1$ must have held since then."; "$<*>\ F$" can be defined as "*true* S $F$."

Below is an example PTLTL property. In this property the goal is to ensure that next is never called on an Iterator without first calling hasNext:

next => (*)hasNext

The event definition should make sure that the call to the hasNext method actually returns true, as well. Recall that (*) means previously, so the property states that the event preceding next must be hasNext.

The original algorithm for PTLTL monitor generation, as outlined in [62, 63], works by using a bitvector to keep the state of each temporal operator in the formula. A series of sequential assignments updates the bitvector as each event arrives. For example, "*hold* S *acquire*" would need one bitvector index to monitor. The assignment for this bitvector index would be "b[0] $\leftarrow$ b[0] and hold or acquire". Fig. 5.8 shows the assignments necessary for each PTLTL temporal operator. Note that if one of the operands to a temporal formula is itself a temporal formula, it will appear as a bitvector index in the assignment. It is, then, essential to generate

the assignments in the proper order (depth-first).

In [100], it was determined that a parallel series of assignments would be more efficient for monitoring PTLTL properties on an FPGA. Sequential assignments are parallelized by back substitution of terms for the bitvector index they computed. This back substitution in an assignment to b[n] is only performed, however, for bitvector indices b[m] that are computed before the assignment to b[n] in the original sequential assignments. For example, consider the following sequential bitvector assignments:

```
b[0]  ←  b[0] or e1
b[1]  ←  b[0] and b[2] or e2
b[2]  ←  e3
```

When parallelized (we us $\leftharpoondown$ to denote parallel assignments and $\leftarrow$ for sequential), the code becomes:

```
b[0]  ⇜  b[0] or e1
b[1]  ⇜  (b[0] or e1) and b[2] or e2
b[2]  ⇜  e3
```

Note how "b[0] or e1" was substituted for "b[0]" in the assignment to b[1] because the assignment to "b[0]" occurred before the assignment to b[1] in the sequential code, while the assignment to b[2] was *not* substituted, because b[2] was computed after b[1] in the sequential code.

By using the parallel assignments it also becomes straightforward to generate an FSM by exhaustively computing and enumerating the reachable bitvectors. This allows us to easily compute the property enable sets of a PTLTL formula by using the algorithm in Fig. 5.3 on the FSM generated from the formula. This is the strategy now used in JavaMOP, while BusMOP continues to use the parallel assignments.

Figs. 5.9(a) and 5.9(b) show the monitor pseudocode for the example above. Fig. 5.9(a) shows the parallel assignment format (which, in this case, is equivalent to the sequential code), while Fig. 5.9(b) shows the FSM output. Note that in the parallel assignments b[0] is initialized to false. In the parallel assignments the output statement outputs the actual category. If it evaluates to false a violation is reported, if it evaluates to true a validation is reported. The FSM output uses aliases for validation and violation because, unlike in LTL, multiple states can be validation or violation states due to the manner in which the FSM is generated (e.g., n1 and n2 are both in the validation alias).

As mentioned in Section 5.2, the enable sets for PTLTL are computed by generating an FSM and using the FSM enable set computation algorithm.

```
b[0] <- hasNext
output(not next or b[0])
```
(a) Parallel Assignments

```
n0[
  default n2
  hasnext -> n1
  next -> n0
]
n1[
  default n2
  hasnext -> n1
  next -> n2
]
n2[
  default n2
  hasnext -> n1
  next -> n0
]
alias violation = n0
alias validation = n1,n2
```
(b) FSM

Figure 5.9: PTLTL Example Output

## 5.4 Linear Temporal Logic with Past

Linear temporal logic (LTL) [102] is often used to specify properties in model checking. LTL formulae allow one to express concepts such as the occurrence of an event requiring that another event happen in the future. Note that runtime monitoring cannot guarantee the correctness of a safety or liveness property. Even though the properties might hold for a given execution of the system, they can only be proved to hold, in general, by exploring every possible state of the program. LTL specifications must be used with this in mind.

Fig. 5.10 shows the complete syntax for LTL supported by our plugin. The operators "*not*", "*and*", "*or*" , and "=>" (implication) are the standard boolean connectives. The operator "[]" stands for "always", meaning the formula following it must hold at all times, while "<>" stands for "eventually", meaning that the formula following it must eventually hold in the future. The operator "o" means "next": the formula following it, say $F$, must hold in the next time step; in terms of MOP, this means that $F$ must hold when the next event occurs. As with PTLTL in the last section, an asterisk in the symbol means the past version, thus "<*>" is "eventually in the past", and "(*)" means previously (next in the past).

⟨*LTL Name*⟩ ::= "*ltl*"
⟨*LTL Syntax*⟩ ::= "*true*" | "*false*"
| ⟨*Event Name*⟩
| "*not*" ⟨*LTL Syntax*⟩
| ⟨*LTL Syntax*⟩"*and*" ⟨*LTL Syntax*⟩
| ⟨*LTL Syntax*⟩"*or*" ⟨*LTL Syntax*⟩
| ⟨*LTL Syntax*⟩"*xor*" ⟨*LTL Syntax*⟩
| ⟨*LTL Syntax*⟩"=>" ⟨*LTL Syntax*⟩
| "[]" ⟨*LTL Syntax*⟩
| "<>" ⟨*LTL Syntax*⟩
| "o" ⟨*LTL Syntax*⟩
| ⟨*LTL Syntax*⟩ "U" ⟨*LTL Syntax*⟩
| ⟨*LTL Syntax*⟩ "∼U" ⟨*LTL Syntax*⟩
| ⟨*LTL Syntax*⟩ "R" ⟨*LTL Syntax*⟩
| "< ∗ >" ⟨*LTL Syntax*⟩
| "(*)" ⟨*LTL Syntax*⟩
| ⟨*LTL Syntax*⟩ "S" ⟨*LTL Syntax*⟩
| ⟨*LTL Syntax*⟩ "∼S" ⟨*LTL Syntax*⟩
⟨*LTL State*⟩ ::= "*violation*" | "*?*"

Figure 5.10: LTL Syntax

The operators "U" and "∼U" are duals of each other; "R" is a preferred alias for "∼U". "U" is "until": "$F_1$ U $F_2$" means "either $F_2$ must hold now, or $F_1$ must hold until $F_2$ eventually holds". "<> $F$" can be defined as "*true* U $F$". The operator "R" means "release". "$F_1$ R $F_2$" means "once $F_1$ holds, $F_2$ can be released in the next time step;" $F_2$ must hold at all periods before $F_1$ holds, and it must hold during the first time step in which $F_1$ holds. "[] $F$" can be defined in terms of "R" as "*false* R $F$". "S" and "∼S" are the past versions of "U" and "R", respectively.

It should be noted that the past time operators in this LTL algorithm do not work exactly like the past time operators available in the PTLTL plugin. This is because the notion of past time operators in that plugin is not compatible with how standard linear temporal logic works. In standard linear temporal logic one essentially is asking "does this formula hold in the first state (i.e. upon receipt of the first event in a trace) of the program?" For example if we use the specification o hasNext, which means next hasNext holds, we are asking if this holds after the first event. For this to hold the second event must be hasNext. This notion of time is static and unshifting, and this is the notion we have adopted for the LTL plugin. The alternative semantics for past time operators of the PTLTL plugin we refer to as shifting time. What this means is that we reevaluate the formula in each state of

the program (i.e. after each event is received). As a clear example if we wrote a formula a S b (a holds since b held, and b held at some point in the past) in the PTLTL plugin, this will be true after the first event if and only if the first event is b, if the second event is a it still holds after the second event. If the third event is c, it ceases to hold again until b occurs again. However, it may hold again.

In the LTL plugin, if one writes the specification a S b, however, it will hold if and only if the first event is b. This is because this notion of time is unshifting and only cares about whether the property holds after the first event. Clearly, at the first event, b could not have occurred in the past, because there is no past. The only event the monitor based on the specification can be aware of is the first event. This means if the first even is a or c it is a violation.

However, if a past time operator is used within a future time construct, the meaning is intuitive. For instance if we monitored the property o(a S b), "b a" and "b b" are valid traces as one would expect. Further for oo(a S b), "b a a" would be a valid trace.

Below is an example LTL property that restates the UNSAFEMAPITER property:

[](useIter => (not updateMap S createColl))

This says that, always, any time we see the useIter event, it must be in the context of not seeing an updateMap event since the creation of the Collection.

The first step of the algorithm is to generate a Buchi automaton from an LTL formula via a translation through two way alternating automata, as presented in [53]. Briefly a two way alternating automata is a six-tuple $A = (Q, \Sigma, \delta, I, F, R)$ where $Q$ is the set of states, is $\Sigma$ the alphabet (in this case $\mathcal{P}_f(\textit{events})$, to allow eventual support for simultaneous events), $\delta = Q \rightarrow \mathcal{P}_f(\mathcal{P}_f(Q \times \Sigma \times Q))$ is the transition function, $I \subseteq \mathcal{P}_f(Q)$ is the initial condition. $F \subseteq Q$ is the set of final states (for finite traces to satisfy the formula) and $R \subseteq Q$ is the set of states that must be repeated (for infinite traces to satisfy the formula). Each tuple as an output of $\delta$ can be thought of as a transition in two directions. The set of states from $Q$ as the first element in the tuple is a conjunction of states that must reach the acceptance condition with an index in the input stepping one event into the past. The set of states from $Q$ as the third element in the tuple is a conjunction of states that much reach the acceptance condition with an index in the input stepping one event into the future. The second element from $\Sigma$ is the set of events which hold that triggers the transition. It is possible for the set of events which holds and the current state to trigger multiple tuples. The resulting top level set is the disjunction

$Q = sub(\varphi) \cup \{\text{END}\}$ *where* $sub(\varphi)$ *are the temporal subformulae of* $\varphi$,
$\Sigma = 2^{\text{AP}}$,
$I = \overline{\varphi}$,
$F = \{\text{END}\}$,
$R$ *is the set of the subformulae in* $sub(\varphi)$ *that are not of the form* $\varphi_1 \,\mathsf{U}\, \varphi_2$
$\delta$ *is defined below (*$\Delta$ *extends* $\delta$ *to* $\mathcal{B}^+(Q)$*):*

$$
\begin{cases}
\begin{aligned}
\delta(\bot) &= \emptyset \\
\delta(\top) &= \{(\emptyset, \Sigma, \emptyset)\} \\
\delta(p) &= \{(\emptyset, \Sigma_p, \emptyset)\} \text{ where } \Sigma_p = \{a \in \Sigma \mid p \in a\} \\
\delta(\neg p) &= \{(\emptyset, \Sigma_{\neg p}, \emptyset)\} \text{ where } \Sigma_{\neg p} = \Sigma \backslash \Sigma_p \\
\delta(\mathsf{X}\,\psi) &= \big\{(\emptyset, \Sigma, e) \mid e \in \overline{\psi}\big\} \\
\delta(\widetilde{\mathsf{X}}\,\psi) &= \big\{(\emptyset, \Sigma, e) \mid e \in \overline{\psi}\big\} \cup \{(\emptyset, \Sigma, \{\text{END}\})\} \\
\delta(\mathsf{Y}\,\psi) &= \big\{(e, \Sigma, \emptyset) \mid e \in \overline{\psi}\big\} \\
\delta(\widetilde{\mathsf{Y}}\,\psi) &= \big\{(e, \Sigma, \emptyset) \mid e \in \overline{\psi}\big\} \cup \{(\{\text{END}\}, \Sigma, \emptyset)\} \\
\delta(\psi_1 \,\mathsf{U}\, \psi_2) &= \Delta(\psi_2) \ \cup (\Delta(\psi_1) \otimes \{(\emptyset, \Sigma, \{\psi_1 \,\mathsf{U}\, \psi_2\})\}) \\
\delta(\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2) &= \Delta(\psi_2) \ \otimes (\Delta(\psi_1) \cup \{(\emptyset, \Sigma, \{\psi_1 \,\widetilde{\mathsf{U}}\, \psi_2\}), (\emptyset, \Sigma, \{\text{END}\})\}) \\
\delta(\psi_1 \,\mathsf{S}\, \psi_2) &= \Delta(\psi_2) \ \cup (\Delta(\psi_1) \otimes \{(\{\psi_1 \,\mathsf{S}\, \psi_2\}, \Sigma, \emptyset)\}) \\
\delta(\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2) &= \Delta(\psi_2) \ \otimes (\Delta(\psi_1) \cup \{(\{\psi_1 \,\widetilde{\mathsf{S}}\, \psi_2\}, \Sigma, \emptyset), (\{\text{END}\}, \Sigma, \emptyset)\}) \\
\delta(\text{END}) &= \emptyset
\end{aligned}
\end{cases}
$$

$$
\begin{cases}
\begin{aligned}
\Delta(\psi) &= \delta(\psi) \text{ if } \psi \text{ is a temporal formula} \\
\Delta(\psi_1 \vee \psi_2) &= \Delta(\psi_1) \cup \Delta(\psi_2) \\
\Delta(\psi_1 \wedge \psi_2) &= \Delta(\psi_1) \otimes \Delta(\psi_2)
\end{aligned}
\end{cases}
$$

Figure 5.11: Conversion from LTL to Two Way Alternating Automaton

*Initialization:* $I'' = \{F\} \times I$, $\nabla = \{F\} \times I$, $T'' = \emptyset$.
*Then, we apply the following saturation procedure for each state* $(X, Y) \in \nabla$ *until we reach a fixed point:*

```
   for each (X', α, Z) ∈ ⨂_{q∈Y} δ(q)
      if X' ⊆ X
(a)   |  if (Y, Z) ∉ ∇ then add (Y, Z) to ∇
      |  if (X, Y, Z, α) ∉ T'' then add (X, Y, Z, α) to T''
      else
      |  for each (F, X, Y, β) ∈ T'' with (F, X) ∈ I''
(b)   |     if (F, X ∪ X') ∉ ∇ then add (F, X ∪ X') to ∇
      |     add (F, X ∪ X') to I''
      |  for each (V, W, X, γ), (W, X, Y, β) ∈ T''
(c)   |     if (W, X ∪ X') ∉ ∇ then add (W, X ∪ X') to ∇
      |     if (V, W, X ∪ X', γ) ∉ T'' then add (V, W, X ∪ X', γ) to T''
```

*Finally, we set* $Q'' = \{X \in 2^Q \mid \exists Y \in 2^Q,\ (X, Y) \in \nabla \text{ or } (Y, X) \in \nabla\}$
$F'' = (2^Q \times 2^F) \cap (Q'' \times Q'')$, *and* $\mathcal{T}'' = \{T''_q \mid q \in Q \backslash R\}$ *where*
$T''_q = \{(\overleftarrow{X}, X, \overrightarrow{X}, \alpha) \in T'' \mid q \notin X \text{ or}$
$\exists (\overleftarrow{Y}, \beta, \overrightarrow{Y}) \in \delta(q) \text{ with } \overleftarrow{Y} \subseteq \overleftarrow{X},\ \overrightarrow{Y} \subseteq \overrightarrow{X},\ \beta \supseteq \alpha \text{ and } q \notin \overrightarrow{Y}\}$

Figure 5.12: Conversion from Two Way Alternating Automaton to Generalized Buchi Automaton

of the conjunctions of states arising from the tuples themselves. This is a more compact representation than the standard format, that allows for more efficient algorithm implementations. The algorithm assumes that the LTL formula is in negative normal form. Our implementation translates to negative normal form and performs several simplifications modulo associativity.

For the sake of completeness Figs. 5.11 and 5.12 show the algorithms for conversion from LTL to two way alternating automata and from two way alternating automata to generalized Buchi automata, respectively. Both of these are taken from [53], but we have the only publicly available implementations. The conversion from generalized Buchi automata to Buchi automata is well known and thus omitted. We use the procedure presented in [42] to convert the Buchi automaton into an FSM suitable for monitoring. Due to the limitations of [42], we only support violations. If one wishes to monitor validation, one may simply monitor for the violation of the negation of the original formula, however, as LTL is invertable. Fig. 5.13 shows the output FSM generated from the formula for UNSAFEMAPITER.

As mentioned in Section 5.2, the property enable sets for LTL formulae are

```
n0[
  createIter -> n0
  default n0
  updateMap -> n0
  useIter -> violation
  createColl -> n1
]
n1[
  createIter -> n1
  default n1
  updateMap -> n0
  useIter -> n1
  createColl -> n1
]
violation[

]
```

Figure 5.13: LTL Example Output

computed by using the algorithm in Fig. 5.3 after a formula has been converted to an FSM.

## 5.5 Chapter Conclusion

This chapter presented the finite state plugins of the MOP framework. The primary goal of these plugins is to increase the efficiency of Runtime Verification. By minimizing the multi-category finite state machines resulting from each plugin, we ensure that a minimum of memory is needed for monitoring finite state properties. The algorithm for computing enable/coenable sets from finite state machines is necessary for the effective use of the optimizations presented in Chapter 3. Likewise, the algorithm for parallel PTLTL assignments improves the circuit size and theoretical number of cycles necessary to make a state transition in the BusMOP system presented in Chapter 4. Lastly, the implementation of an LTL plugin with past increases the expresssivity of Runtime Verification, as it is the first such monitoring algorithm in existence. All of the experiments in Section 3.4.2 of Chapter 3.

# Chapter 6

# Context-Free Grammars

## 6.1 Chapter Introduction

Context-free grammars (CFG) are nearly as widely adopted by the average programmer as are regular expressions. Numerous context-free parser generators such as Bison [21] exist and are widely used. CFGs offer a level of expressibility greater than that of finite-monitor logics, and allow for the specification of properties that involve proper nesting and a notion of counting.[1]

Runtime Verification (RV) is a relatively new formal analysis approach in which specifications of requirements are given together with the code to check, as in traditional formal verification, but the code is checked against its requirements at runtime, as in testing. A large number of Runtime Verification approaches and systems, including TemporalRover [44], JPaX [59], JavaMaC [82], Hawk/Eagle [41], Tracematches [6,10], J-Lo [23], PQL [89], PTQL [54], MOP [32,33], Pal [29], RuleR [17], etc., have been developed recently. In a Runtime Verification system, monitoring code is generated from the specified properties and integrated with the system one wishes to monitor. Therefore, a Runtime Verification approach consists of at least three interrelated aspects: (1) a specification formalism, used to state properties to monitor, (2) a monitor synthesis algorithm, and (3) a program instrumentor. The chosen specification formalism determines the expressivity of the Runtime Verification approach and/or system.

Monitoring safety properties is arbitrarily complex [109]. Recent developments in Runtime Verification, however, show that regular and temporal-logic-based formal specifications can be efficiently monitored against large programs. As shown by a series of experiments in the context of Tracematches [10] and JavaMOP [33], parametric regular and temporal logic specifications can be monitored against large

[1] Work on monitoring context-free grammars was performed with Dongyun Jin, Chen Feng, and Grigore Roşu. It was first published in [91] with an extended version in [92]. The current GLR-based algorithm used in MOP was adapted by Dennis Griffith.

programs with little runtime overhead, on the order of 12% or lower. However, both regular expressions and temporal logics reduce to ordinary finite automata when monitored, so they have inherently limited expressivity. More specifically, most Runtime Verification approaches and systems consider only *flat execution traces*, or execution traces without any structure. Consequently, users of such Runtime Verification systems are prevented from specifying and checking *structured properties*, those properties referring to the program structure such as properties with requirements on the contents of the program call stack. Examples of such structured safety properties include "a resource should be released in the same method which acquired it" or "a resource cannot be accessed if the unsafe method foo is in the current call stack".

### 6.1.1 Example

An important and desirable category of properties that cannot be expressed using regular patterns is one in which pairs of events need to match each other, potentially in a nested way. For example, suppose that one prefers to use one's own locking mechanism for thread synchronization. As usual, for multiple reasons including the allowance of re-entrant synchronized methods (in particular to support recursion), locks are allowed to be acquired and released multiple times by any given thread. However, the lock is effectively released, so that other threads can acquire it, only when the lock releases match the lock acquires. One may want to impose an even stronger locking safety policy: the lock releases should match the lock acquires within the boundaries of each method call. This property is vacuously satisfied when locks are acquired and released in a structured manner using synchronized blocks or methods, like in Java 4+, but it may be easily violated when one imple-

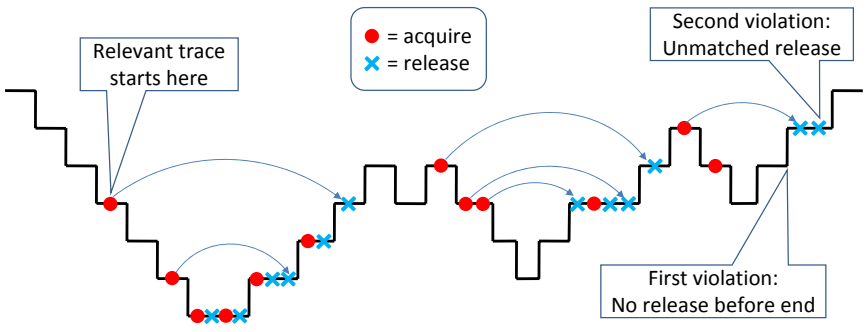



Figure 6.1: Example trace for structured acquire and release of locks.

ments one's own locking mechanism or uses the semaphores available in Java 5. For example, Fig. 6.1 shows an execution violating this basic safety policy twice (each deeper level symbolizes a nested method invocation). First, the policy is violated when one returns from the last (nested) method invocation because one does not release the acquired lock. Second, the policy is also violated immediately after the return from the last method invocation because the lock is released twice by its caller, but acquired only once.

Supposing that the system is instrumented to emit events begin and end when methods of interest are started and terminated, and that the events acquire and release are triggered when the lock of interest is acquired and released, respectively, then here is an initial, straightforward way to express this safety policy as a context-free grammar:

$$S \quad \rightarrow \quad \epsilon \mid S \text{ begin } S \text{ end} \mid S \text{ acquire } S \text{ release}$$

Because of the production $S \rightarrow \epsilon$, the pattern is able to terminate ($\epsilon$ is the empty trace). This pattern will match any trace with begin events in balance with end events because these two events occur only in the production $S \rightarrow S$ begin $S$ end,[2] where they are matched. The $S$ at the beginning of the production allows an unbounded number of these balanced groupings in a row, e.g., begin begin end end begin end is a valid trace with two balanced groupings in a row. The production $S \rightarrow S$ acquire $S$ release is similar to that with begin and end events, allowing balanced groupings of acquire and release events. Because all the productions have the same recursive symbol, $S$, it is possible for begin/end pairs to nest within acquire/release pairs, and vice versa. Thus a valid trace would be begin acquire begin end release end begin acquire acquire release release end.

This pattern is simple and works, however, it has a deficiency in that it must monitor every begin of every method in a given program, even those which do not perform any thread synchronization. Next, we present a more efficient grammar that ignores begin events that happen before the first acquire event in a trace.[3] The next grammar looks complicated, but we must stress that it is only complicated to improve monitoring efficiency.

[2] Note that $S \rightarrow a \mid b$ is shorthand for $S \rightarrow a$, $S \rightarrow b$.

[3] Events that occur before the first event in a valid trace are ignored by the monitoring algorithm. Events which begin a valid trace are known as *creation events* [33].

$$
\begin{aligned}
S &\rightarrow \epsilon \mid S \text{ acquire } M \text{ release } A \\
M &\rightarrow \epsilon \mid M \text{ begin } M \text{ end} \mid M \text{ acquire } M \text{ release} \\
A &\rightarrow \epsilon \mid A \text{ begin} \mid A \text{ end}
\end{aligned}
$$

```
perthread SafeLock(Lock l) {
  event acquire before(Lock l) :
        call(* Lock.acquire()) && target(l) {}
  event release after(Lock l) :
        call(* Lock.release()) && target(l) {}
  event begin before() :
        execution(* *.*(..)) && !within(Lock) {}
  event end after() :
        execution(* *.*(..)) && !within(Lock) {}

  lr_lazy : S -> epsilon | S acquire M release A,
            M -> epsilon | M begin M end
               | M acquire M release,
            A -> epsilon | A begin | A end

  @fail {
     System.out.println("Unsafe lock operation found!");
  }
}
```

Figure 6.2: JavaMOP specification for the SAFELOCK safety property using the CFG plugin.

Again, the productions begin with recursive references to $S$ to allow for repetition of balanced groupings. This time, however, the $S$ productions only allow for acquire and release, not begin and end. This ensures that only begins and ends occurring after the first acquire are monitored. The non-terminal $M$ stands for "matched" sub-traces, i.e., traces in which all the pairs begin/end and acquire/release are properly matched, and $A$ stands for sequences of (not necessarily matched) begin and end events. The $A$ productions are necessary because failures would be reported for end events at the end of a trace due to the lack of a matching begin that occurred before the acquire creation event. The $A$ productions also allow for more begin events after the last release because we do not want method calls after the last release to cause the invocation of the failure handler.

Any (finished or unfinished) execution trace that is not a prefix of a word in the language of $S$ in the context-free grammar (CFG) above is an execution that

violates the safety policy. The CFG Runtime Verification technique presented in this chapter and implemented as a logic-plugin in JavaMOP is able to monitor safety properties expressed as CFGs like above. Monitoring-Oriented Programming (MOP) and JavaMOP (the Java implementation of MOP) are discussed in Chapters 2 and Chapter 3.

Fig. 6.2 shows this SAFELOCK property expressed as a JavaMOP specification, using the CFG logic plugin. The modifier perthread tells JavaMOP to consider events from separate threads as separate traces. This is particularly important as we do not wish begins and ends of separate threads to cause the pattern to fail. SAFELOCK denotes the name of the specification, while the list after SAFELOCK is the list of parameters to the specification (see below). SAFELOCK is parametric in the Lock because we do not wish the releases and acquires of separate Locks to interfere. The keyword event introduces an event; the event is first given a name, and then its trigger is defined using an AspectJ [79] advice (before the colon) and a pointcut (after the colon). Of particular note, however, is !within(Lock), used so that we do not monitor the begins and ends of the acquire and release methods, which would cause the pattern to fail. JavaMOP's generic approach to parametric specifications is described in [35] and [31]. Because of this generic approach, the logical formalisms in which properties are expressed need not be aware of the parameters; parameters are added automatically and generically by the JavaMOP framework.

The keyword lr_lazy introduces the CFG pattern. Three other possible keywords could be used: lr, lalr, and lalr_lazy.[4] The two lazy keywords mean that when an event is encountered that causes a pattern match failure, the failure handler is invoked, but the event itself is not kept in the monitor state so that more failures can be found. If the error causing event were kept, as in the non-lazy keywords, each following event would cause an error, regardless of whether it should. The lazy method is how most programming language parsers work, allowing multiple syntax errors to be caught in one parse. lr and lalr determine which table generation algorithm is used (see Section 6.2 for more information on the two table generation algorithms). The first nonterminal in the pattern is assumed to be the start symbol of the grammar (see Section 6.2.1). Lastly, @fail introduces a *pattern failure handler*. The code within the braces following @fail runs whenever the pattern fails to match because an invalid event for a given point in a trace is seen. As an

[4]The current GLR plugin accepts only "cfg" as a keyword, we plan to again allow a "lazy" option, however. This discussion is maintained for completeness.

alternative, JavaMOP allows @match handlers. This gives extra power to our CFG plugin because context-free languages are not closed under complementation.

The code generated automatically from the JavaMOP specification in Fig. 6.2, following the technique described in the rest of the chapter, has more than 700 lines of (human unreadable) AspectJ code. We ran this property against a hand-crafted program, which generated the sequence of events seen in Fig. 6.1. Both pattern failures were successfully caught in a single run because the CFG plugin does not add failure inducing events to the monitor state when lr_lazy is used. If the keyword lr is used instead of lr_lazy, only the first failure is caught by the generated monitor.

### 6.1.2 Chapter Contributions

Several approaches have been proposed to monitor context-free properties. For example, Program Query Language (PQL) [89] is based on a description language that encompasses the intersection of context-free languages. Hawk/Eagle [41] uses a fix-point logic and RuleR [17] uses a rule based logic that can specify context-free properties. These approaches propose what we feel are rather complex solutions for monitoring parametric context-free patterns. They generate inefficient monitoring code in many cases, thus preventing practical parametric context-free property monitoring with these systems. The inefficiency of PQL in comparison to JavaMOP with context-free patterns is discussed in Section 6.3. This chapter shows that monitoring (the LR(1) and LALR(1) subsets of) parametric context-free patterns is *practical*.[5] We generate *non-parametric* monitors instead of parsers for the defined context-free pattern. Parameters are handled separately, using the algorithm in [31, 35]. This way, we provide an efficient system for monitoring parametric context-free properties. Our algorithm is totally different from the monitoring algorithm used by the PQL system [89], which mixes the handling of parameters and monitoring of context-free patterns.

When monitoring pattern languages, such as extended regular expressions (ERE) or CFG, we wish to report a match anytime a trace at a given point in program execution matches the pattern. For example, if we have a pattern that is looking for writes to a closed file, we might use the ERE close write write*. We wish to report a match on every write, so that we can locate all of the trouble spots in the program. We call this matching every good prefix of the trace because close

[5]The GLR plugin currently used by MOP can monitor any context-free grammar, and is almost as efficient as the plugin used for the results in this Chapter.

write is a prefix of close write write which is a prefix of close write write write, and we wish for a match to be reported on each of these prefix traces. We provide two methods to deal with the problem of monitoring good prefixes. One is to modify the LR(1) parsing algorithm with a *stack copying* process. The second method, called *guaranteed acceptance*, was discovered after our work in [91].

We also performed an extensive evaluation of the CFG monitoring algorithm using the DaCapo [22] benchmark suite and properties used previously to evaluate Runtime Verification systems [26, 33]. The properties are expressed as CFGs in this evaluation rather than regular expressions for use in JavaMOP. Even when monitored using the CFG plugin, however, these regular pattern based specifications still use constant space. We thus performed an evaluation of three strictly context-free properties – which use theoretically unbounded space – to show that, even with such properties, the overhead is reasonable, and to show the usefulness of context-free properties. The results of this analysis compare favorably with PQL and Tracematches, two state-of-the-art runtime monitoring systems. One of these properties (IMPROVEDLEAKINGSYNC) is expressible in neither PQL nor Tracematches, for reasons explained in Section 6.3. Another of the properties (SAFEFILEWRITER), while expressible in PQL, is not expressible in Tracematches because Tracematches has limited ability to express structured properties, rather than the full generality of the LR(1) languages.

Over both the adapted regular properties and the new strictly context-free properties, the overhead of JavaMOP with CFGs is, on average, over 8 times less than Tracematches on properties that Tracematches is able to express, and over 12 times less than PQL on properties that can be expressed in PQL. On all but 9 of the 45 benchmark/property pairs that generated events,[6] the overhead is less than 5% in JavaMOP with CFGs.

Contributions exclusive to [92], include three more versions of the original LR(1)-based cfg plugin. LALR(1) was added and a version of both LR(1) and LALR(1) that stay in an error state when an error token is encountered (using la/lalr instead of lr/lalr_lazy). The GLR mode added by Dennis Griffith, however, is described in [93]. Also presented in this chapter is the concept of guaranteed acceptance (see Section 6.2.2). Lastly, from [93], we include the formalization of (co)enable set computation from CFGs.

[6]Overall there are 66 benchmark/property pairs, but 21 of them generate no events, and are removed to more fairly represent the overhead of runtime monitoring.

### 6.1.3 Chapter Outline

The remainder of the chapter is as follows: Section 6.2 explains our CFG monitor synthesis technique in JavaMOP, including considerations for suffix matching and enable set computation; Section 6.3 explains our experimental setup and the results of our experiments; Section 6.4 illustrates related work; Section 6.5 concludes the chapter.

## 6.2 Context-Free Patterns in JavaMOP

We support the LR(1) subset of context-free grammars (CFGs), as well as LALR(1) which is a subset of LR(1). LR(1) is so named because it parses input **L**eft to right and produces a **R**ight-most derivation. The 1 denotes that one token of look-ahead is used. The LA in LALR stands for look-ahead, because, under certain conditions,states in the LR(1) table with different look-aheads may be merged in the LALR(1) table (see Section 6.2.2).

LR(1) can only recognize a subset of the deterministic context-free languages, which are themselves a strict subset of the context-free languages (CFLs) [4, 66]. LR(1), however, is an expressive subset, able to define the syntaxes of most modern programming languages. We chose LR rather than LL because LR recognizes a larger number of grammars without translation. We base our implementation on the Knuth algorithm [83] for LR(1) parser table generation as presented in [4]. While the "action" and "goto" tables generated are normal LR(1) "action" and "goto" tables, the algorithm used to parse has been modified to work in the context of monitoring, explained in detail below. We added a plugin, which generates LALR(1) using the algorithm presented in [4], because LALR(1) generates smaller tables in some cases. The LALR(1) tables are, at worst, identical to the LR(1) tables; they are never larger. The downside of LALR(1), however, is that it is a strict subset of LR(1). Comparisons between the table size of LR(1) and LALR(1) for the properties we tested can be found in Section 6.3, and an explanation of the LALR(1) optimization can be found in Section 6.2.2. Sections 5.1–5.2.3 cover standard issues related to LR parsing from a monitoring context, while the remainder of Section 6.2 covers new issues specific to adapting LR parsing to monitoring.

### 6.2.1 Preliminaries

A CFG $G$ is defined as a tuple of the form, $G = (NT, \Sigma, P, S)$. $\Sigma$, the alphabet of the CFG, is often referred to as the set of terminals. A very special terminal, \$, represents the *end* of the input. $NT$ is the set of nonterminals. $P$ is the set of productions, which define what strings nonterminals derive. $NT \cup \Sigma$ is often called the set of symbols. Productions have the form $A \rightarrow \gamma$, where $A \in NT$ and $\gamma$ is a string that either consists of symbols, or is the empty string, $\epsilon$, i.e. $\gamma \in (\Sigma \cup NT)^*$. We use the conventional alternation operator, "|": a production of the form $A \rightarrow \gamma_0|\gamma_1$ can be equivalently represented as two productions $A \rightarrow \gamma_0$ and $A \rightarrow \gamma_1$. $S$ is the start symbol – that non-terminal from which all strings in the language are derived. For example, $G = (\{A\}, \{a, b\}, P_0, A)$ where $P_0 = \{A \rightarrow aAb|\epsilon\}$ is a simple CFG for the language $\{a^n b^n | n \in \mathbb{N}\}$. The non-terminal $A$ can derive $aAb$ an indeterminate amount of times before deriving $\epsilon$, allowing $a^n b^n$ for any $n \in \mathbb{N}$.

Two important sets are defined for every non-terminal in a grammar: the ***first*** and *follow* sets. These are used in "action" table construction. The ***first*** set will be used to decide which terminals in the given grammar define monitor creation events (we shall be more specific about this below). The *follow* set will be useful in illustrating the fundamental challenge of monitoring CFGs. The ***first*** set of a non-terminal $A$, denoted ***first(A)***, is the set of all terminals $t$ such that the sub-strings which reduce to $A$ may possibly begin with $t$. The follow set, denoted ***follow(A)***, is the set of terminals which follow the strings which reduce to $A$. These terminals signify a reduction by $A$.

A reduction is the step whereby a right hand side of a production, $\gamma$, is replaced by the left hand side non-terminal in the production. For example, if we have the string $aaabbb$, and we are using our example grammar, $G$ , we can perform a reduction with $\gamma = \epsilon$ resulting in $aaaAbbb$. We can then perform another reduction with $\gamma = aAb$ resulting in $aaAbb$, eventually we reach $aAb$, which reduces to $A$ (and because $A$ is the start symbol, $aaabbb$ must be in $L(G)$, where $L(G)$ represents the language derived by $G$).

### 6.2.2 CFG Monitoring Algorithms

We developed the CFG monitoring algorithms based on existing parsing algorithms. *Action* and *goto* tables are generated from the given CFG pattern and used to advance the monitor at runtime according to the observed events. Moreover, the CFG monitor is unaware of relevant parameters since they are handled by the

| | Action | | | | | Goto | |
|---|---|---|---|---|---|---|---|
| | $ | acquire | release | begin | end | | S |
| 0 | shift(error) | shift(14) | shift(error) | shift(16) | shift(error) | 0 | 9 |
| 1 | reduce(S, 2) | reduce(S, 2) | reduce(S, 2) | reduce(S, 2) | reduce(S, 2) | 1 | error |
| 2 | reduce(S, 2) | reduce(S, 2) | reduce(S, 2) | reduce(S, 2) | reduce(S, 2) | 2 | error |
| 3 | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | 3 | error |
| 4 | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | 4 | error |
| 5 | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | 5 | error |
| 6 | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | reduce(S, 3) | 6 | error |
| 7 | reduce(S, 4) | reduce(S, 4) | reduce(S, 4) | reduce(S, 4) | reduce(S, 4) | 7 | error |
| 8 | reduce(S, 4) | reduce(S, 4) | reduce(S, 4) | reduce(S, 4) | reduce(S, 4) | 8 | error |
| 9 | accept | shift(15) | shift(error) | shift(17) | shift(error) | 9 | error |
| 10 | shift(error) | shift(15) | shift(3) | shift(17) | shift(error) | 10 | error |
| 11 | shift(error) | shift(15) | shift(error) | shift(17) | shift(4) | 11 | error |
| 12 | shift(error) | shift(15) | shift(7) | shift(17) | shift(error) | 12 | error |
| 13 | shift(error) | shift(15) | shift(error) | shift(17) | shift(8) | 13 | error |
| 14 | shift(error) | shift(14) | shift(1) | shift(16) | shift(error) | 14 | 10 |
| 15 | shift(error) | shift(14) | shift(5) | shift(16) | shift(error) | 15 | 12 |
| 16 | shift(error) | shift(14) | shift(error) | shift(16) | shift(2) | 16 | 11 |
| 17 | shift(error) | shift(14) | shift(error) | shift(16) | shift(6) | 17 | 13 |

Figure 6.3: LALR(1) Tables for SAFELOCK

underlying MOP framework. This greatly simplifies the monitoring algorithm. We next introduce the monitor algorithms in more detail.

### CFG Simplification

The CFG plugin first applies some standard simplifications to the given grammar [66]. The first step of simplification is the removal of non-generating nonterminals ($A$ is non-generating if $\forall s \in \Sigma^*$, $s$ cannot be reduced to $A$ in one or more reductions). The next step in the simplification process is the removal of nonterminals which are unreachable from the start symbol ($A$ is unreachable from the start symbol if there is no string $\gamma$ that contains $A$ and reduces to the start symbol in any number of steps). The last step removes $\epsilon$-productions from the grammar. After $\epsilon$-productions have been removed from $G$, resulting in $G'$, $L(G') = L(G) - \epsilon$.[7] A monitor matching the empty event trace has little utility, so we feel this is a fair compromise.

[7] The remaining productions are restructured to account for the removal of $\epsilon$-productions without changing the recognized language, other than as mentioned.

```
1   globals action_table, goto_table
2   initialize stack.push(initial_state)
3   procedure monitor(event, stack)
4     locals state, state', stack', A
5     state ← stack.top()
6     while (true) {
7       switch (action_table[state,event].action_type) {
8       case shift :
9         state' ← action_table[state, event].next_state
10        if (state' = error) {
11          pattern failure
12          break while
13        }
14        stack.push(state')
15        if (action_table[state', $].action_type = reduce) {
16          stack' ← stack.copy()
17          monitor($, stack')
18        }
19        break while
20      case reduce :
21        stack.pop(action_table[state, event].pop)
22        A ← action_table[state, event].non_terminal
23        state' ← stack.top()
24        stack.push(goto_table[state', A])
25        break switch
26      case accept :
27        pattern match
28        break while
29      }
30    }
```

Figure 6.4: CFG Monitoring Algorithm with Stack Copying.

### Tables and Monitoring

After simplification, the tables are generated for a deterministic push-down automaton (DPDA), that is, a deterministic finite automaton with a stack, which is to be used as a monitor. The algorithm first adds a production to introduce $\$$. If the start symbol of the original grammar was $S$, it adds a production $S' \rightarrow S\$$. The next step is generating the *canonical LR(1) (or LALR(1)) collection*. The collection consists of *collection items*; each item represents a state in the automaton. A collection item is a set of productions with a marker, *, for the current position in the right hand side, augmented with a look-ahead. For example, a possible collection item for the simple HasNext pattern $S \rightarrow next\ next$ is $[\{S' \rightarrow * \ S\ \$, \$\}, \{S \rightarrow * \ next\ next, \$\}]$, another is $[\{S \rightarrow next\ *\ next, \$\}]$. In both of these collection items, the look-ahead is $\$$. The collection item is first created for the start production, $S' \rightarrow S\$$. All productions for $S$ are added to the state with * at the beginning of the right hand side, as can be seen in our example collection item. If there is a production for $S$ such that the first symbol is a nonterminal, $A$, all the productions of $A$ will also be added, with * at the beginning. This process is transitively closed. The next collection items are generated by advancing * in the production, and then

taking the transitive closure for any nonterminals immediately following *. Once the collection is created, the tables are generated by treating each item as a state. The productions in the item are considered for each alphabet symbol (including $). If the marker appears in front of said terminal, a *shift* action is generated. The algorithm decides which collection item to shift to by looking for the collection item where there is a production with the same right hand side as the production that caused the shift action, but with * advanced one position. For example, the next collection from $[\{S \rightarrow next \ * \ next, \$\}]$ would be $[\{S \rightarrow next\ next *, \$\}]$. *Shift* actions can never be generated for $; the algorithm disallows it. The handling of *shift* actions by the parsing algorithm can be seen below. If, however, there is a production in the item such as $[\{S \rightarrow next\ next *, \$\}]$, where the marker appears at the end, and the terminal in question is the look-ahead, a *reduction* action is generated. More explanation of this algorithm can be found in [4].

When a new event arrives, the monitoring algorithm must decide how to modify the stack. The tables are given in a generic intermediate form, which is converted by the Java shell into two Java arrays.

Pseudo-code for our monitoring algorithm is given in Fig. 6.4. The significance of lines 15–18 is explained Section 6.2.2. An entry in *action_table* specifies, via the *action_type*, the type of action: *shift*, *reduce*, or *accept*. Each type of action also requires additional information in order for the algorithm to process said action. An entry in *goto_table* simply identifies the next state for the DPDA. Fig. 6.3 shows a parse table as it would be used by the algorithm. This is the LALR(1) table for the first SAFELOCK grammar given in Section 6.1. We show the LALR(1) table because the LR(1) table has 50 states, and the tables can be used interchangeably by the algorithm.

The *shift* action entry contains the next state for the DPDA in the *next_state* field (in parentheses in the shift actions in Fig. 6.3). A *shift* action simply pushes the next state on the stack, if the next state is *not* the error state (lines 10–14). If, however, the table indicates that the next state *is* the error state, the algorithm reports a pattern fail and breaks *without* touching the stack (lines 11–12). This allows the algorithm to continue to find more pattern failures. After a successful *shift* action, the while loop is broken, allowing execution of the monitored program to continue until the next relevant event (line 19).

The *reduce* action is more complicated. The field *non_terminal* describes which non-terminal ($A$) the production $A \rightarrow \gamma$ reduces to (the first field in the reduce actions Fig. 6.3), while the field *pop* denotes how many states to pop from the stack

($|\gamma|$) (the second field in the reduce actions in Fig. 6.3). The reduction proceeds by popping the specified number of states from the stack and consulting *goto_table* to decide the next state (lines 20–25). The state used for indexing *goto_table* is not the current state, but rather the state at the top of the stack after the specified number of states has been popped (line 21). An indeterminate number of reductions can happen in a row, but there *must* be *shift* at the end of the reduction sequence before the algorithm can terminate for a given event. The reductions happen before the *shift* to simulate the look-ahead of one token specified by the 1 in LR(1).

The *accept* action, which directs the DPDA to signal a pattern match, has no special fields, as no more information is necessary (lines 26–28).

### The LALR(1) Optimization

LALR(1) table generation is a standard modification of the LR(1) table generation algorithm [4]. LALR(1) tables are constructed the same way as LR(1) tables, save that states corresponding to collection items with the same *core* are merged as they are discovered. The core of a collection item is that item with the look-aheads removed. This means that LALR(1) tables can be no larger than LR(1) tables, but may be considerably smaller.

However, this process may result in reduce-reduce conflicts (states where a reduction to two or more different nonterminals with the same look-ahead may occur) that do not exist when the LR(1) construction method is used. Shift-reduce conflicts (states where a shift with a given token, $a$, and a reduction with $a$ as the look-ahead are possible) cannot be introduced because they require that there be two productions in a given collection item such that one has the position marker in front of a terminal $t$ (meaning that $t$ should be shifted), while another production has the marker at the end of the production, with $t$ as the look-ahead (meaning that we should reduce to the left hand side when $t$ is seen). Obviously the conflict must occur *before* merging, because if the shift inducing production were in a state $s_0$, while the reduce inducing production where in $s_1$, $s_0$ and $s_1$ would have different cores, and not be merged. To see how reduce-reduce actions may be introduced, consider the two collection items: $[\{A \rightarrow a, \$\}, \{B \rightarrow a, a\}]$ and $[\{A \rightarrow a, a\}, \{B \rightarrow a, \$\}]$ Before merger, no conflict exists. Looking at the first collection item, there is no conflict between the two productions because it says to reduce to $A$ only when the look-ahead is \$, and $B$ only when the look-ahead is $a$. The collection item after merger, however, is: $[\{A \rightarrow a, \$\}, \{B \rightarrow$

$a, a\}, \{A \to a, a\}, \{B \to a, \$\}]$, which contains two reduce-reduce conflicts (one on look-ahead $a$, the other on $\$$). The only way to check if such a conflict is introduced by the LALR(1) merger is to generate LR(1) tables to see if there is still a reduce-reduce conflict.

### Handling the End of Trace

The (LA)LR(1) algorithm assumes that the string of terminals to be evaluated is *completely known ahead of time*. Thus, it knows where the end of the string (denoted as \$) is. This is important because some reductions happen with the \$ symbol as the look-ahead, and the accept action can *only* be recognized when the next input is \$. To be consistent with our notion of monitoring, it must be possible to consider the trace prefix at a given point in a run of a program as an *entire* trace. The algorithm must then assume $\$$ after every event.

Our implementation of the algorithm attempts to reduce with \$ as the look-ahead after *every* valid *shift* (lines 15–18). The problem with reducing with $\$$ as the look-ahead where possible is that all state of the current trace evaluation is lost. This means that the monitor could only accept the minimal trace that matches the CFG pattern if no special care were taken.

Since our notion of monitoring reports pattern matches for every current trace that matches the pattern,[8] one possible option is to *copy* the stack before we perform any reductions with \$ as the look-ahead.

This copying ensures that the stack is intact for the next, and subsequent events, allowing for multiple pattern matches. For example, consider the language denoted by the regular expression $ab^*$. While we would suggest using the ERE plugin for such a language, it is a clear example to illustrate the effect of copying. With no copying the algorithm would accept for only $a$. Because it popped during the pattern match phase, if it sees a $b$ it will report failure. With copying it will report success for $a$, and then success again on the input of $b$, and for any subsequent input of $b$. An important optimization to copying is to only copy the stack if there is a reduction with \$ as the look-ahead, rather than blindly for every *shift* operation. This optimization will not help $ab^*$, but it will help for many other languages. In fact, for $\{a^n b^n | n \in \mathbb{N}\}$, only one copy is necessary no matter how long the input. Any grammar accepting unbounded repetition at the end of the pattern (like $ab^*$), will require copying on each input of the repeated character.

[8] This is irrespective of suffix matching which actually generates multiple monitors.

$$
\begin{array}{rcl}
\mathrm{G}(\epsilon) & = & \{\emptyset\} \\
\mathrm{G}(t) & = & \{\{t\}\} \\
\mathrm{G}(A) & = & \bigcup_{A \to \beta} \mathrm{G}(\beta) \\
\mathrm{G}(\beta_1\beta_2) & = & \{S \cup T \mid S \in \mathrm{G}(\beta_1), T \in \mathrm{G}(\beta_2)\} \\
\mathrm{P}(\gamma) & = & \{S \cup T \mid A \to \beta_1\gamma\beta_2, S \in \mathrm{P}(A), T \in \mathrm{G}(\beta_1)\} \\
\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e) & = & \mathrm{P}(e)
\end{array}
$$

Figure 6.5: CFG $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ Defining Equations

Our experience with stack copying led us to an important observation: when there is a reduction with \$ as the look-ahead, acceptance is *guaranteed*. That is to say, there is never a situation in which there is a reduction with \$ as the look-ahead that results in a parse failure. This is a consequence of the parse table generation algorithm, and Section 6.2.5 covers the correctness of this notion, which we refer to as *guaranteed acceptance*. Using guaranteed acceptance to accept whenever a reduction with \$ as the look-ahead is possible is another alternative for matching all good prefixes of a pattern. Guaranteed acceptance is always correct. We maintain the discussion on stack copying because the proof of the stack copying algorithm is easier to understand, and because it is an interesting, though less practical, alternative to guaranteed acceptance.

### 6.2.3 (Co) Enable Set Generation

To find the enables sets of a CFG we find the least fixed point of the equations in Fig. 6.5. Here, informally, $\mathrm{G}(A)$ is the set of events generated by the CFG, if the symbol $A$ were used as the start symbol of the CFG. The rule $\mathrm{G}(\beta_1\beta_2) = \{S \cup T \mid S \in \mathrm{G}(\beta_1), T \in \mathrm{G}(\beta_2)\}$ generalizes this notion to entire strings of symbols. P is the enable sets function generalized to strings that include both non-terminals and terminals. For example, the prefixes of $abTB$ for $B$ would be $\{\{a, b\} \cup S \mid S \in G(T)\}$. For a rule, $A \to \beta_1 B \beta_2$, $\mathrm{P}(B)$ needs to cope with the fact that $A$ has its own enables set of possible prefixes. Thus its definition unions possible prefixes of $A$ with the sets of symbols that are generated by $\beta_1$. The rest of MOP only needs to know sets of prefixes for events, thus $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ is just the restriction of P to events. To find coenable sets, we use these same equations after reversing the right-hand side of productions.

### 6.2.4 CFG-plugin Implementation

The CFG plugin allows the user to specify a number of events and a CFG specifying allowable event traces. The events become the terminals of the CFG, i.e., $\Sigma$. The translation steps from specification to working Java code gradually transform the specification into AspectJ join points (the events) and aspects (the synthesized monitors), which are then woven into the original application using any off-the-shelf AspectJ compiler.

As described in Section 3.4 of Chapter 6, several features are needed for monitors to support optimized suffix matching.

The first is identification of monitor creation events.[9] As already mentioned, monitor creation events are events which, when encountered as the *first* event in a trace, would not lead to an immediate failure. For CFGs this would imply an event that can begin a sub-string which reduces to the start symbol. This is the same as the definition of *first* set as given earlier. Thus, the monitor creation events for the CFG plugin are those events which are in *first(S)*, where $S$ is the start symbol for the grammar.

Additionally, it is necessary to define a hash encoding for CFG based monitors because our suffix matching algorithm uses a HashSet to find monitors with *potentially* equivalent states quickly. We decided that two simple defining aspects of CFG based monitors are stack depth and the current state of the monitor (the top of the stack). We chose to xor them together (a broadly used operation for combining two binary quantities into one quantity representative of the two in the same number of bits) because the hashCode method must return an integer. Lastly, we need an equality method (to resolve collisions) defining when two CFG based monitors have *actually* equivalent states. Two CFG monitors can only be equal iff they have the same stack contents. It will be fairly rare for two proper CFG monitors to be equivalent, as they do not have finite state like the other logic plugins of MOP. Thus, it is important for failed equality testing to be fast. Because of this, we check to see if two monitors have the same stack depth before beginning element-wise comparisons.

### 6.2.5 Proofs of Correctness

We next prove the correctness of the proposed CFG monitoring algorithms.

[9] Though, as mentioned, this is also necessary for total matching.

```
1  globals stack, ip, action_table, goto_table
2  initialize stack.push(initial_state), ip ← 0
3  procedure parse()
4    locals state, state', a, A
5    while (true) {
6      state ← stack.top()
7      a ← get_token_at(ip)
8      switch (at[state, a].action_type) {
9      case shift :
10       state' ← action_table[state, a].next_state
11       if (state' = error) {
12         report_error
13         advance ip
14         continue while
15       }
16       stack.push(state')
17       advance ip
18       continue while
19     case reduce :
20       stack.pop(action_table[state, a].pop)
21       A ← action_table[state, a].non_nonterminal
22       state' ← stack.top()
23       stack.push(goto_table[state', A])
24       continue while
25     case accept :
26       accept
27       return
28     }
29   }
```

Figure 6.6: ASU Algorithm.

First, we prove the online monitoring algorithm for CFG using stack copying correct. We achieve this by showing that our algorithm detects pattern failures and pattern matches of the observed trace in the same way as the ASU parsing algorithm [4], as given in Fig. 6.6.

***Theorem****: For every finite prefix of a (possibly infinite) program trace*[10] *and a CFG pattern, the MOP algorithm will notify a failure of the pattern if the ASU algorithm would notify a parse failure due to a bad token, and pattern match if ASU would notify success, given that prefix as total input.*

***Proof:*** First, we construct a new parsing algorithm, as shown in Fig. 6.7. This new algorithm can be proved equivalent to the one in Fig. 6.6 as follows. The major difference between these two algorithms is that the pointer (*ip*) increment is moved to the outer loop in Fig. 6.7. This change does not affect the behavior of the algorithm:

1. For a shift action, both algorithms carry out the same operation except that Fig. 6.6 increases the pointer and continues to the next action, while Fig. 6.7 breaks the inner loop, increases the pointer in the outer loop, and then

[10]Each prefix is an $\mathcal{E}$-trace at a given point in a program as per Definition 1 in Chapter 3.

```
1  globals stack, ip, action_table, goto_table
2  initialize stack.push(initial_state), ip ← 0
3  procedure parse()
4    locals state, state', a, A
5    while (true) {
6      a ← get_token_at(ip)
7      while (true) {
8        state ← stack.top()
9        switch (at[state, a].action_type) {
10       case shift :
11         state' ← action_table[state, a].next_state
12         if (state' = error) {
13           report_error
14           break while
15         }
16         stack.push(state')
17         break while
18       case reduce :
19         stack.pop(action_table[state, a].pop)
20         A ← action_table[state, a].non_nonterminal
21         state' ← stack.top()
22         stack.push(goto_table[state', A])
23         continue while
24       case accept :
25         accept
26         return
27       }
28     }
29     advance ip
30   }
```

Figure 6.7: Modified ASU Algorithm.

continues to the next action. Both are equivalent.

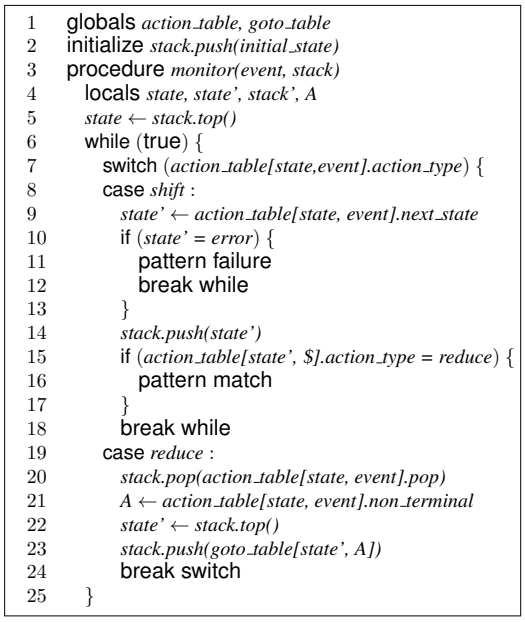
2. For reduction, Fig. 6.7 chooses to stay in the inner loop, which is identical to Fig. 6.6, and continues the loop without increasing the pointer.

3. For acceptance (pattern match in monitors), both algorithms are identical.

Now we can prove the correctness of the monitoring algorithm in Fig. 6.4 by comparing it with the modified parsing algorithm in Fig. 6.7.

The major difference distinguishing the monitoring algorithm from ASU is that the former has to wait for the next event extracted from the execution of the monitored program while the latter can actively retrieve the next token, which is handled in the outer loop in Fig. 6.7. Therefore, we only need to prove that the *monitor* procedure in Fig. 6.4 produces the same result as the inner loop in Fig. 6.7, given the same state and event to process.

It is straightforward to compare both pieces of code: the only difference between them is the *stack copying* (lines 14–17) in Fig. 6.4. It is needed because we wish to continue parsing *after an accept*, and because we can never actually see $ as an event. We copy the stack after a shift and check for actions with $ as



```
1   globals action_table, goto_table
2   initialize stack.push(initial_state)
3   procedure monitor(event, stack)
4     locals state, state', stack', A
5     state ← stack.top()
6     while (true) {
7       switch (action_table[state,event].action_type) {
8       case shift :
9         state' ← action_table[state, event].next_state
10        if (state' = error) {
11          pattern failure
12          break while
13        }
14        stack.push(state')
15        if (action_table[state', $].action_type = reduce) {
16          pattern match
17        }
18        break while
19      case reduce :
20        stack.pop(action_table[state, event].pop)
21        A ← action_table[state, event].non_terminal
22        state' ← stack.top()
23        stack.push(goto_table[state', A])
24        break switch
25    }
```

Figure 6.8: CFG Monitoring Algorithm with Guaranteed Acceptance.

the input. The only actions possible on this recursive call are reduce and accept because $ can never be shifted.[11] Due to this, the recursion is always bounded at depth one. This is the major difference between the MOP and ASU algorithms. Because $ can never be an event, we must speculatively guess the end of input after every symbol seen. The recursive call must happen iff there is a valid reduction with $ as the look-ahead. Because the algorithm repeats until a shift action, error, or accept happens, we ensure that, if the recursive call happens, it must happen after the processing of each input.[12] Cloning the stack allows us to reduce and accept, while still maintaining the original stack to continue monitoring events as if the end of input had *not* been seen. Thus, this change is equivalent to the ASU algorithm in terms of language recognition because both possibilities (the arrival or non-arrival of $) are explored. That is, the MOP algorithm will report accept for a given prefix if ASU would, given that prefix as its total input, and it, additionally, retains enough state to continue parsing future (longer) traces. Violation is handled identically in both algorithms. ■

[11] This is a property of the CFG parsing table generation algorithm, which we use without proof. It is obvious, however, because $ is not a part of the original grammar.

[12] Accept need not be considered because it can only happen when the input is $, which only occurs during a recursive call.

We next prove the correctness of our online monitoring algorithm for CFG using the notion of *guaranteed acceptance*. It is achieved by showing that a reduction with $ as the look-ahead must result in accept with $ as look-ahead[13] in one or more reductions. Fig. 6.8 shows the algorithm modified to take advantage of guaranteed acceptance. Note that there is no longer an accept case because accept is discovered in the shift case, on lines 15–16.

***Theorem****: If the action table entry for a given state specifies reduction with* $ *as look-ahead, the stack copying processes must lead to an acceptance. That is to say, in one or more reductions with* $ *as the look-ahead, an accept action must occur with* $ *as the look-ahead.*

***Proof:*** From the canonical LR(1) construction algorithm [4], we know that a given non-terminal $B$ can only be reduced to with look-ahead terminal $a$, if there exists some production $C \rightarrow \gamma_0 B a \gamma_1$ or a sequence of productions $C \rightarrow \gamma_0 B_0 a \gamma_1$, $B_0 \rightarrow \gamma_2 B_1$, $B_1 \rightarrow \gamma_3 B_2$, ... $B_n \rightarrow \gamma_{n+2} B$. Note that the sequence may contain cycles of one or more production, such as a production $C \rightarrow bC$, but there must be a finite amount of "cycle reductions" because there is a finite amount of input, and the CFG simplification we perform removes non-generating nonterminals.[14]

In the algorithm, $ is treated as a special terminal that exists in only one production, the start production $S' \rightarrow S\$$, where $S$ is the original start symbol of the grammar. This production is added by the algorithm and is the only existence of $ in the grammar. Intuitively, when the contents of the parse stack *correspond* to $S$, the algorithm accepts. This means that if we can reach a stack containing only state corresponding to $S$ through one or more reductions with $ as the look-ahead, we can accept. The original ASU algorithm and our version of the algorithm with stack copying perform all of these reductions.

As a result of the two facts above, a reduction with $ as look-ahead, for non-terminal $A$, can only occur in the table if there is some sequence of productions $S \rightarrow A_0\$$, $A_0 \rightarrow \gamma_0 A_1$, $A_1 \rightarrow \gamma_1 A_2$, ... $A_n \rightarrow \gamma_n A$, or the production $S \rightarrow A\$$. If we have $S \rightarrow A\$$, then we can obviously accept on the reduction to $A$ because this will result in a stack with only contents corresponding to $A$ ($A = S$ from above), which is the accept condition, or there will be some finite number of cycles with non-terminal $A$ that eventually leads to a stack containing only contents corresponding to $A$ because there must be a finite amount of cycle reductions. If,

[13] Note that this is the only look-ahead ever possible for accept.

[14] It is clear that only non-generating nonterminals could have infinite cycles, because the table generation algorithm works bottom up, and chooses shift over reduce on conflict.

however, we have the sequence of productions, we can still accept because there is a sequence of reductions from $A$ to $S \rightarrow A_0\$$ given by the sequence of productions. Again, even if the sequence contains cycles, eventually acceptance must be reached because there must be a finite amount of cycle reductions. ■

## 6.3 Evaluation

We evaluated the JavaMOP CFG plugin and compared its performance to PQL and Tracematches on the DaCapo benchmark suite [22]. We used the LR(1) tables, with the lazy algorithm, and suffix matching. We feel this gives a worst case, while giving the most fair comparison to Tracematches, which uses suffix matching semantics. LR(1) tables are of greater or equal size, so they cannot be faster than the LALR(1) tables. The lazy mode has the potential to be slower because it continues to modify the stack after a failure, while the normal mode does not. Suffix matching is obviously slower than total trace matching because it creates one total match monitor for every suffix of the trace.

### 6.3.1 Experimental Settings

Our experiments were carried out on a machine with 1.5GB RAM and Pentium 4 2.66GHz processor. The operating system used was Ubuntu Linux 7.10. JavaMOP 1.0 was used, so the memory numbers in particular are not as good as they are in the current version of JavaMOP (3.0), as they predate both the enable set and coenable set optimizations. We used the DaCapo benchmark version 2006-10; it contains eleven open source programs [22]: antlr, bloat, chart, eclipse, fop, hsqldb, jython, luindex, lusearch, pmd, and xalan. The provided default input was used together with the -converge option to execute the benchmark multiple times until the execution time falls within a coefficient of variation of 3%. The average execution time of six iterations after convergence are then used to compute the runtime overhead. Therefore, Fig. 6.9 percentages should be read "$\pm 3$" (meaning negative numbers are possible).

### 6.3.2 Properties

The following general properties borrowed from [26] were checked in the evaluation:

- HASHMAP: An object's hash code should not be changed when the object is a key in a HASHMAP;
- HASNEXT: For a given iterator, the hasNext() method should be called between all calls to next();
- SAFEITERATOR: Do not update a Collection when using the Iterator interface to iterate its elements.

We also defined three new properties to showcase the power of the CFG plugin; they are all properly context-free:

- IMPROVEDLEAKINGSYNC: The original LeakingSync specified in [26] *only* allows synchronized accesses to synchronized collections. This causes spurious failures because the synchronized methods call the unsynchronized versions. Our improved version allows calls to the unsynchronized methods so long as they happen within synchronized calls.
- SAFEFILEINPUTSTREAM: SAFEFILEINPUTSTREAM is a modification of our SAFELOCK property from Fig. 6.2. It ensures that a FileInputStream is closed in the same method in which it is created.
- SAFEFILEWRITER: SAFEFILEWRITER ensures that all writes to a FileWriter happen between creation and close of the FileWriter, and that the creation and close events are matched pairs.

More properties have been checked in our experiments; we chose the first three regular-language-based properties (HASHMAP, HASNEXT, and SAFEITERATOR) to include in this chapter because they generate a comparatively larger runtime overhead. We excluded those with little overhead in JavaMOP. For every property, we provide overhead percentages for JavaMOP, as well as PQL and Tracematches where possible. We run the JavaMOP monitors in suffix matching mode; the decentralized indexing of monitors was used in all the experiments (see [33]). We chose the AspectJ compiler 1.5.3 (AJC) in the evaluation to compile the JavaMOP generated monitoring AspectJ code. Guaranteed acceptance and stack copying have the same performance on each of these patterns because none of the properties is able to generate more than one pattern match from a given parameter instance, meaning that the number of stack copies is very minimal. Additionally, the properties have been written in a method that ensures minimal stack size (such

as using left recursion instead of right recursion). For Tracematches we used the most recent published version from [121].

| | HASHMAP | | | HASNEXT | | | SAFEITERATOR | | |
|---|---|---|---|---|---|---|---|---|---|
| | MOP | PQL | TM | MOP | PQL | TM | MOP | PQL | TM |
| antlr | 3 | 6 | 0 | 1 | 2 | 3 | 2 | 82 | 0 |
| bloat | 14 | 9 | -2 | 1112 | 5929 | 2452 | 627 | 8694 | 11258 |
| chart | -1 | 1 | -1 | -1 | 3 | 0 | 2 | 50 | 11 |
| eclipse | 0 | 1 | 1 | 0 | 2 | -1 | -2 | 1 | 2 |
| fop | 3 | 2 | 0 | 0 | 2 | -1 | -1 | 24 | 5 |
| hsqldb | 0 | 3 | 15 | 0 | 6 | 15 | 0 | 78 | 17 |
| jython | 0 | 23 | 15 | 0 | 0 | 13 | 0 | 12 | 16 |
| luindex | 1 | 8 | 1 | -2 | 93 | 2 | 3 | 181 | 9 |
| lusearch | 1 | 1 | 8 | -1 | 59 | 9 | 4 | 132 | 34 |
| pmd | -1 | 0 | 3 | 191 | 1870 | 52 | 178 | 1334 | 175 |
| xalan | 0 | 5 | 1 | 0 | 0 | 2 | 1 | 53 | 10 |

| | IMPROVEDLEAKINGSYNC | | | SAFEFILEINPUTSTREAM | | | SAFEFILEWRITER | | |
|---|---|---|---|---|---|---|---|---|---|
| | MOP | PQL | TM | MOP | PQL | TM | MOP | PQL | TM |
| antlr | 1 | N/E | N/E | 3 | 113 | -1 | 2 | 22 | N/E |
| bloat | 13 | N/E | N/E | 1 | 128 | 0 | 27 | 97 | N/E |
| chart | 4 | N/E | N/E | 0 | 29 | 1 | 0 | 37 | N/E |
| eclipse | 1 | N/E | N/E | -2 | 3 | 0 | -2 | 1 | N/E |
| fop | 1 | N/E | N/E | -2 | 58 | -1 | -2 | 47 | N/E |
| hsqldb | 1 | N/E | N/E | 1 | 280 | 21 | 2 | 95 | N/E |
| jython | 41 | N/E | N/E | 0 | 937 | 12 | 1 | crashes | N/E |
| luindex | 1 | N/E | N/E | -1 | 233 | 6 | 0 | 33 | N/E |
| lusearch | 2 | N/E | N/E | -1 | 137 | 7 | 0 | 49 | N/E |
| pmd | 36 | N/E | N/E | -1 | 547 | 1 | -2 | 658 | N/E |
| xalan | 3 | N/E | N/E | -1 | 90 | 3 | -2 | 164 | N/E |

Figure 6.9: Average percent runtime overhead for JavaMOP CFG (MOP), PQL, and Tracematches (TM) (convergence within 3%); N/E means "not expressible".

### 6.3.3 Results

Fig. 6.9 shows the percent overheads of JavaMOP using the CFG plugin, PQL, and Tracematches. N/E refers to specifications that were not expressible. Negative numbers can be attributed both to the 3% noise in the measurements and instruction cache layout changes due to the weaving process. Tracematches is unable to support IMPROVEDLEAKINGSYNC because the property is truly context-free. PQL is also unable to support it because it requires events corresponding to the beginning and end of synchronized method calls, and PQL can only trigger events on the end of method calls. Tracematches cannot support SAFEFILEWRITER because it is a

| Property | HASHMAP | HASNEXT | SAFEITERATOR |
|---|---|---|---|
| antlr | 0 | 0 | 1990 |
| bloat | 361519 | 143103032 | 75944328 |
| chart | 8773 | 6819 | 569345 |
| eclipse | 20888 | 3252 | 32759 |
| fop | 17265 | 281 | 49959 |
| hsqldb | 0 | 0 | 0 |
| jython | 443 | 106 | 177554 |
| luindex | 9615 | 28140 | 82162 |
| lusearch | 416 | 0 | 405428 |
| pmd | 11354 | 33294563 | 25476563 |
| xalan | 124155 | 0 | 1009649 |

| | IMPROVEDLEAKINGSYNC | SAFEFILEINPUTSTREAM | SAFEFILEWRITER |
|---|---|---|---|
| antlr | 8472 | 0 | 0 |
| bloat | 5587905 | 259 | 385 |
| chart | 634260 | 0 | 0 |
| eclipse | 74630 | 930 | 0 |
| fop | 182407 | 12 | 0 |
| hsqldb | 0 | 0 | 0 |
| jython | 23969673 | 544 | 0 |
| luindex | 1559386 | 1114 | 0 |
| lusearch | 1291992 | 0 | 0 |
| pmd | 26291289 | 10 | 32 |
| xalan | 5146036 | 13604 | 0 |

Figure 6.10: Number of events handled by JavaMOP

pure context-free specification. However, Tracematches *can* support SAFEFILEINPUTSTREAM because it has the ability to access call stack depth via the cflowdepth pointcut term, which is provided only by the ABC compiler for AspectJ.

Over one run of the entire DaCapo benchmark suite, more than 355 *million* events (Fig. 6.10) were triggered. Tracematches has the same number of events throughout the tests because it uses the same instrumentation technique as JavaMOP. We had no good method to obtain the number of events generated in PQL; we assume it was less because PQL performs a static optimization which removes unnecessary optimization points. It is interesting to note that in the cases of HasNext with antlr, lusearch, and xalan that there are no events, despite the fact that these three benchmarks have events for SAFEITERATOR. The reason for this is that the SAFEITERATOR instruments Collection.remove, so it is possible for SAFEITERATOR to have events in programs with no actual Iterators.

The average overhead of JavaMOP over 45 program/property pairs that actually generate events is 50%. There are two considerations here, however: (1) we chose specifically those properties that generated the largest overheads (HASNEXT and SAFEITERATOR in bloat), (2) when the two largest overheads are removed, the average over the remaining 43 pairs drops to a very reasonable 12%. Further, the average JavaMOP overhead for properties expressible in PQL that generated events was 61% over 36 pairs, while PQL's overhead on these same properties was 583%. Similarly, for Tracematches expressible properties that generated events, JavaMOP's overhead was 64% over 33 pairs, while Tracematches was 414%. Tracematches, PQL, and JavaMOP all feature the same two pairs which have extremely large overhead compared to the median (HASNEXT and SAFEITERATOR in bloat). When these two pairs are removed from the three averages, the average overhead for JavaMOP with respect to PQL expressible properties is 12%, while PQL still weighs in at 199%. Tracematches is comparable to JavaMOP, with JavaMOP and Tracematches both at 12%. Since Tracematches does not support the full generality of (deterministic) context-free grammars, we view comparable performance to Tracematches as favorable to our approach, especially given that, in the overall data set, our average overhead is over 8 times lower than Tracematches' overhead.

The largest overheads seen, across all three systems, are for SAFEITERATOR and HASNEXT in bloat. This is due to bloat's extensive use of iterators. Bloat is a bytecode optimizer, which uses iterators to process bytecode. PQL and Tracematches perform worse on SAFEITERATOR than they do on HasNext, while our performance is the opposite. The reason for this is that HasNext creates a

| Property | Original | HASHMAP | HASNEXT | SAFEITERATOR |
|---|---|---|---|---|
| antlr | 2.3 / 10.1 | 2.0 / 10.6 | 1.8 / 10.6 | 2.0 / 10.8 |
| bloat | 5.6 / 8.9 | 6.9 / 8.9 | 5.9 / 8.7 | 541.0 / 10.6 |
| chart | 20.1 / 11.3 | 20.8 / 11.4 | 17.0 / 11.3 | 20.7 / 11.5 |
| eclipse | 27.0 / 22.1 | 30.7 / 22.2 | 27.4 / 22.1 | 28.6 / 22.3 |
| fop | 12.3 / 9.1 | 13.2 / 9.2 | 10.9 / 9.0 | 10.2 / 9.1 |
| hsqldb | 80.8 / 7.6 | 80.2 / 7.6 | 76.4 / 7.5 | 77.5 / 7.6 |
| jython | 3.9 / 19.0 | 4.1 / 19.0 | 3.8 / 19.0 | 3.9 / 19.1 |
| luindex | 4.2 / 6.9 | 4.0 / 7.0 | 4.6 / 6.9 | 4.7 / 7.1 |
| lusearch | 5.2 / 6.2 | 5.2 / 6.3 | 5.7 / 6.2 | 5.3 / 6.3 |
| pmd | 22.0 / 8.6 | 22.3 / 8.7 | 24.0 / 8.6 | 888.1 / 8.9 |
| xalan | 21.7 / 10.2 | 23.8 / 10.5 | 26.2 / 10.2 | 29.1 / 10.3 |

| Property | Original | IMPROVEDLEAKINGSYNC | SAFEFILEINPUTSTREAM | SAFEFILEWRITER |
|---|---|---|---|---|
| antlr | 2.3 / 10.1 | 2.1 / 10.7 | 2.4 / 10.7 | 2.2 / 10.7 |
| bloat | 5.6 / 8.9 | 7.9 / 10.0 | 5.0 / 8.9 | 5.6 / 8.9 |
| chart | 20.1 / 11.3 | 17.0 / 11.3 | 17.8 / 11.3 | 16.4 / 11.3 |
| eclipse | 27.0 / 22.1 | 28.9 / 22.1 | 30.7 / 22.1 | 27.1 / 22.1 |
| fop | 12.3 / 9.1 | 14.6 / 9.2 | 11.9 / 9.0 | 12.0 / 9.0 |
| hsqldb | 80.8 / 7.6 | 87.2 / 7.5 | 78.2 / 7.5 | 79.3 / 7.5 |
| jython | 3.9 / 19.0 | 4.0 / 19.2 | 4.0 / 19.1 | 3.6 / 19.1 |
| luindex | 4.2 / 6.9 | 5.6 / 7.0 | 4.2 / 6.9 | 4.6 / 6.9 |
| lusearch | 5.2 / 6.2 | 5.8 / 6.4 | 5.6 / 6.3 | 5.7 / 6.3 |
| pmd | 22.0 / 8.6 | 22.2 / 8.8 | 24.2 / 8.6 | 22.9 / 8.6 |
| xalan | 21.7 / 10.2 | 24.4 / 10.3 | 22.0 / 10.3 | 26.5 / 10.2 |

Figure 6.11: Maximum memory usage in MB (Maximum Heap Memory Usage) / (Maximum Non-Heap Memory Usage).

far larger number of monitors in JavaMOP because it creates a monitor for every call to next, while SAFEITERATOR only creates monitors on a call to create. The pattern for SAFEITERATOR, however, is more complex. This shows that JavaMOP has, relatively speaking, more overhead in generating and handling the monitor set for suffix matching than it does in matching the pattern, while PQL and Tracematches overheads are more affected by the complexity of the pattern. Note that JavaMOP with CFGs far outperforms both PQL and Tracematches on these 2 program/property pairs.

SAFEFILEINPUTSTREAM is an interesting case: it is required to match the begin and end of methods. Instrumenting the begin and end of *every* method would be atrociously slow, however. We perform a static analysis which finds those methods in which FileInputStream's are actually used. Then, we instrument only those methods for begin and end. Because Tracematches, also, is pointcut based, we are able to perform the same optimization for Tracematches, so the numbers shown are with the optimization enabled. PQL is not pointcut based so the optimization cannot be applied; however, the PQL property does not match begins and ends of methods (recall: PQL can only match the ends), so this is not an issue. In PQL we specify SAFEFILEINPUTSTREAM by using an interesting PQL-specific feature called within. The idea of within is that a property matches only *within* a given method or methods matching a particular pattern (in the case of SAFEFILEINPUTSTREAM we use the pattern _._ which specifies all methods of all classes). Additionally, PQL will only instrument the same methods that JavaMOP and Tracematches instrument because within only instruments methods which can generate relevant events.

The memory overhead is reasonable in our experiments: overall, it is 33% on average with a 4% median (see Fig. 6.11 for a pair-wise breakdown). There are two extreme cases of memory overhead caused by JavaMOP monitors: bloat-SAFEITERATOR and pmd-SAFEITERATOR. Our investigation shows that both programs, bloat and pmd, make intensive use of vectors, and create numerous iterators to do computation over the vectors throughout the execution. Note that every creation of the iterator leads to the creation of a monitor instance for SAFEITERATOR using our technique. Hence, a huge number of monitor instances were created in these two benchmarks. While the iterator object is usually used in a small scope and then released, the vectors are not released until the end of the execution, preventing the removal of the created monitor instances. In other words, all the monitor instances created during the execution of bloat and pmd were kept

| | HASHMAP | HASNEXT | SAFEITERATOR |
|---|---|---|---|
| LR(1) states | 6 | 5 | 15 |
| LALR(1) states | 6 | 5 | 15 |
| | IMPROVEDLEAKINGSYNC | SAFEFILEINPUTSTREAM | SAFEFILEWRITER |
| LR(1) states | 19 | 20 | 18 |
| LALR(1) states | 15 | 8 | 11 |

Figure 6.12: Comparison of LR(1) and LALR(1) tables.

alive until the execution ended, resulting in the observed massive memory usage. On the contrary, we can see that a large number of monitors were also created for bloat-HASNEXT and pmd-HASNEXT but with much less memory overhead. HASNEXT has one monitor created for every iterator object, and when the iterator is released, the corresponding monitor will also be removed. Since most iterators were released shortly after creation, only a few monitors existed at the same time during the execution resulting in much lower memory overhead. Compared with the results of Tracematches [10], we believe there is still some room for improvement with regard to memory usage in our approach. The memory overhead of our approach does not cause unnecessary loss of performance during the evaluation, indicating that it is not a bottleneck for the efficiency of monitoring.

Fig. 6.12 shows a comparison of parse table size between LR(1) and LALR(1) in terms of number of states. No reduce-reduce conflicts were introduced in any of our properties, and the space savings can be significant. Although, in either case, the tables are small enough that the size difference has no effect on runtime. It is interesting to note that the three regular language properties see no savings from LALR(1).

Our experiments with guaranteed acceptance found no benefit to the patterns we tested because, as mentioned, each pattern can only accept once per parameter instance, and the stack depth is kept to a minimum by using careful grammar design. Not all properties, however, can or should be written in such a way that only one acceptance can be generated per parameter instance. In extreme cases, such as patterns that feature unbounded repetition at the end of the pattern, such as $a^*$, guaranteed acceptance can provide a *lower asymptotic complexity*. Consider the grammar $S \rightarrow \epsilon|aS$ which monitors $a^*$. For each $a$ that arrives, with guaranteed acceptance a *constant time*[15] step of adding $a$ to the stack and accepting occurs.

[15]It is constant except when the stack size must be grown.

With stack copying, the stack must be copied on each arrival of $a$. This copying, however, will take time linear in the amount of times $a$ has been previously seen. Thus, with guaranteed acceptance, the monitoring time is linear in the number of $a$ events in the program run while stack copying is quadratic.

## 6.4 Chapter Related Work

Work related to context-free grammars in MOP can be divided into two main categories: context-free or more powerful logics in runtime monitoring, and context-free grammars in testing and verification.

### 6.4.1 Runtime Monitoring

Many approaches have been proposed to monitor program execution against formally specified properties. Interested readers can refer to [33] for an extensive discussion on existing runtime monitoring approaches. Briefly, all runtime monitoring approaches except MOP [30, 32, 33] have their specification formalisms hardwired and *only two of them* share the same logic (LTL). A thorough discussion of the MOP framework can be found in Chapter 2. This observation strengthens our belief underlying MOP — there probably is *no silver-bullet* specification formalism for all purposes. Also, most approaches focus on detecting either violations (pattern failures in CFG) or validations (pattern matches in CFG) of the desired property and support only fixed types of monitors, e.g., online monitors that run together with the monitored program or offline monitors that check the logged execution trace after program termination.

Specifically, there are four orthogonal attributes of a runtime monitoring system: logic, scope, running mode, and handlers. The logic specifies which formalism is used to specify the property. The scope determines where to check the property; it can be class invariant, global, interface, etc. The running mode denotes where the monitoring code runs; it can be inline (weaved into the code), online (operating at the same time as the program), outline (receiving events from the program remotely, e.g., over a socket), or offline (checking logged event traces).[16] The handlers specify what actions to perform under exceptional conditions; there can be violation and validation handlers. It is worth noting that for many logics, violation

[16] Offline implies outline, and inline implies online.

| Approach | Logic | Scope | Mode | Handler |
|---|---|---|---|---|
| JPaX [59] | LTL | class | offline | violation |
| TemporalRover [44] | MiTL | class | inline | violation |
| JavaMaC [82] | PastLTL | class | outline | violation |
| Hawk [41] | Eagle | global | inline | violation |
| RuleR [17] | RuleR | global | inline | violation |
| Tracematches [10] | Reg. Exp. | global | inline | validation |
| J-Lo [23] | LTL | global | inline | violation |
| Pal [29] | modified Blast | global | inline | validation |
| PQL [89] | PQL | global | inline | validation |
| PTQL [54] | SQL | global | outline | validation |

Figure 6.13: A Selection of Monitoring Systems

and validation are not complementary to each other, i.e., the violation of a formula does not always imply the validation of the negation of the formula.

Most runtime monitoring approaches can be framed in terms of these attributes, as illustrated in Fig. 6.13. For example, JPaX can be regarded as an approach that uses linear temporal logic (LTL) to specify class-scoped properties, whose monitors work in offline mode and only detect violation. In general, JavaMOP has proven to be the most efficient of the runtime monitoring systems despite being generic in logical formalism.

Of the systems mentioned in Fig. 6.13, only PQL [89], Hawk/Eagle [41], and RuleR [17] can handle arbitrary context-free properties. Hawk/Eagle adopts a fix-point logic and uses term rewriting during the monitoring, making it rather inefficient. It also has problems with large programs because it does not garbage collect the objects used in monitoring. In addition, Hawk/Eagle is not publicly available.[17] Because of this and the fact that Hawk/Eagle has not been run on DaCapo [22] with the same properties, we cannot compare our CFG plugin with Hawk/Eagle. RulerR is a simplification of Eagle that is rule based rather than $\mu$-calculus based, but it still has the ability to specify context-free properties. The current implementation is not built for efficiency or ease of expression with regards to context-free properties. In addition to PQL, we decided to perform comparisons with Tracematches [10], as it is able to monitor a very limited set of context-free properties using compiler-specific support provided by their special AspectJ compiler, ABC [9], and because it is a very efficient system. Pal [29] is able

[17] [10] makes an argument for the inefficiency of Hawk/Eagle. Since Hawk/Eagle is not publicly available (only its rewrite based algorithm is public [41]), the authors of Hawk/Eagle kindly agreed to monitor some of the simple properties from [26]. We have confirmed the inefficiency claims of [10] with the authors of Hawk/Eagle.

to monitor properties that take calls and returns into account, giving a limited context-free ability for this one case. Pal is implemented for C, rather than Java, and the implementation is not publicly available.

### 6.4.2 Context-free Grammars in Testing and Verification

Context-free grammars have seen use in several areas of testing and verification not immediately related to runtime monitoring.

Attributed context-free grammars were used as a means to generated test input and output pairs by [46]. The generated test inputs and outputs could be used both the test the specification from which the test grammar was designed, as well as the final implementation of a specification, using automatic test drivers. Their test case generator was capable of generating test cases from the grammar both randomly and systematically. The attributes of the context-free test generation grammars allow a user to attach context sensitive information to parts of the grammar, and allow for refinement of test case generation in order to avoid redundant test cases. Earlier attempts of test case generation via grammars [56, 68, 104] were employed to generate test input only for compilers and parsers rather than programs and their specifications (though, [45] used grammars to generate test cases in much the same way, it could not generate outputs, and it generated far too many similar test cases). In [90] context-free grammars were used to generate test data for VLSI (very large scale integration) circuits. [114] applied the concept of test case generation using context-free grammars to Java virtual machine implementations.

All of these approaches differ quite a bit from runtime monitoring. The overhead of these approaches is not nearly so important because they are used to generate test cases in an offline manner, rather than running at the same time as a program that is under testing or a production system, situations for which runtime monitoring is intended. Additionally, runtime monitoring attempts to monitor behavior of a system rather than to generate test cases.

[71] used context-free grammars for an entirely different purpose that is more related to runtime monitoring than is test case generation. They created an interface specification language that uses context-free grammars to provide stub code for model checking. The grammar specifies the sequence of method invocations allowed by the component. The stubs are called by the code under model checking, providing a means of modular model checking. The grammar generated stubs execute during the model checking process to ensure that the non-stub portions

of code always follow the specification of method calls given by the grammar. In this way it is similar to runtime monitoring with context-free grammars, as the grammar is used to specify intended behavior, and flag errors when the behavior is not followed at runtime. Our work differs primarily in that it is designed to enforce behavior in a running system rather than to abstract a component, and in that it is parametric, whereas the interface grammars are not.

## 6.5 Chapter Conclusion

We implemented a CFG logic plugin for JavaMOP using a modified LR(1) parsing algorithm. We also implemented an optimization of table generation, which uses the LALR(1) state merging technique, leading to smaller tables, but may result in extra reduce-reduce conflicts. Our first modification to the algorithm is based on the novel idea of *copying* the stack in order to "predict" a possible reduction with $ (end of string) as a look-ahead without destroying the state of the monitor. An important optimization and simplification possibility is *guaranteed acceptance*, wherein the algorithm accepts when a reduction with $ as look-ahead is possible; this saves the copying operation, which can take arbitrarily long to perform, since the stack is unbounded. We showed, empirically, that our algorithm is efficient and faster than the state-of-the-art for monitoring CFG properties.

# Chapter 7

# String Rewriting Systems

## 7.1 Chapter Introduction

This chapter presents efficient monitoring for parametric string rewriting systems (SRS). String rewriting systems are Turing-complete, allowing the formal specification of any conceivable safety property. Unfortunately, (co)enable sets for SRS cannot be computed (the problem reduces to the halting problem).[1]

Runtime Verification (RV) is a formal analysis approach in which specifications of requirements are given together with the code to check, as in traditional formal verification, but the code is checked against its requirements at runtime, as in testing. A large number of runtime verification techniques and systems, including TemporalRover [44], JPaX [59], JavaMaC [82], Hawk/Eagle [41], Tracematches [6, 10], J-Lo [23], PQL [89], PTQL [54], MOP [32, 33], Pal [29], RuleR [17], etc., have been developed recently, and the overall approach has gained enough traction to spawn its own conference [14]. In a Runtime Verification system, monitoring code is generated from the specified properties and integrated with the system to monitor. Therefore, a Runtime Verification approach consists of at least three interrelated aspects: (1) a specification formalism, used to state properties to monitor, (2) a monitor synthesis algorithm, and (3) a means to instrument programs. The chosen specification formalism determines the expressivity of the Runtime Verification approach and/or system.

Monitoring safety properties is arbitrarily complex [109]. Early developments in Runtime Verification, showed that *parametric* regular and temporal-logic-based formal specifications can be efficiently monitored against large programs. A parametric monitor associates monitor states with different object instantiations for the given parameters. This allows for specification of properties about the relationships of objects, e.g., a relationship between a Collection object and its associated

[1] Work on monitoring string-rewriting systems was performed with Grigore Roşu. It was originally presented in [94].

Iterator objects in Java.[2] As shown by experiments with Tracematches [10] and the most recent experiments using JavaMOP [76], parametric regular and temporal logic specifications can be monitored against large programs with little runtime overhead, on the order of 15% or lower.

However, both regular expressions and temporal logics are monitored using finite automata, so they have inherently limited expressivity. More specifically, most Runtime Verification approaches and systems consider only *flat execution traces*, or execution traces without any structure. Consequently, users of such Runtime Verification systems are prevented from specifying and checking *structured properties*, those properties referring to the program structure such as properties with requirements on the contents of the program call stack. PQL [89], Hawk/Eagle [41], and RuleR [17] provide more expressive logics, but these are relatively inefficient [6, 10, 33]. More recently, JavaMOP was extended to support efficient context-free monitors with runtime overheads very similar to the earlier finite-state logics [92]. While this work allows for checking many structured properties, it does not have the full power to specify any possible safety property. In this chapter, we introduce an algorithm for monitoring parametric deterministic string rewriting systems, to serve as an efficient Runtime Verification technique for specifying and monitoring arbitrarily complex properties; indeed, string rewrite systems are known to be as expressive as Turing machines [28]. We also provide an implementation of our algorithm as an MOP *logic plugin* [32, 33], so it can be used as integral part of the JavaMOP Runtime Verification system. By abuse of vocabulary, we will refer to deterministic string rewriting systems as string rewriting systems and abbreviate them SRSs.

### 7.1.1 Examples

Safety properties that require more expressivity than a context-free language are generally more intimately related to the specifics of the program under verification/test than those that may be monitored using context-free or finite logics. Conversely, less specific properties, such as correct API usage, tend to be finite state [86]. As a relatively simple, and admittedly contrived, example of a non context-free property, consider the Java class, RandomEquality defined in Fig. 7.1. The idea of this class is to provide a random string of numbers from the set

[2] Typestates [118] a popular concept in software engineering and software analysis, can be monitored with parametric monitors that have only one parameter.

```
public class RandomEquality {
  int numberOfNumbers;

  public RandomEquality(int numberOfNumbers){
    this.numberOfNumbers = numberOfNumbers;
  }

  public int nextNumber() {
    return genNextNumber(numberOfNumbers--);
  }

  public boolean hasNextNumber(){
    return numberOfNumbers > 0;
  }

  private int genNextNumber(int currentNumber){
    //some logic that may or may not be correct
  }
}
```

Figure 7.1: A Java class that provides random number sequences of any length that maintain equality

$\{0, 1, 2\}$ of a given length defined by the parameter passed to the constructor that maintains equality, that is, that the number of 2's is equal to the number of 1's is equal to the number of 0's.

The JavaMOP specification presented in Fig. 7.2, which uses the new srs logic plugin, is able to catch any failures of this class to provide equality. JavaMOP specifications begin with a declaration of the name of the specification and parameters. Here the property is named EQUALITYCHECK, and one parameter re of type RandomEquality. The parameters allow us to associate separate monitor states with each object instantiation of the parameters. In this case, with one parameter, there will be one monitor state associated with each object instance of RandomEquality in the program under test. This is important because we would not want calls to different object instances of the RandomEquality class to interfere with each other as such would assuredly lead to false positives and negatives.

The next part of a JavaMOP specification is the declaration of events. Here we are able to generate four different events: done, e0, e1, and e2. The events are defined using a superset [93] of AspectJ [79] advice with embedded pointcuts. Here, the event done is defined to occur when the hasNextNumber() method

is called and returns false, signaling the end of the randomly generated number string.[3] The events e0, e1, and e2 all correspond to calls of nextNumber() where the proper number in $\{0, 1, 2\}$ is returned.

[3]Note that this requires properly calling hasNextNumber() before calling hasNextNumber(). This can be ensured in JavaMOP using a different, finite state, property.

```
EqualityCheck(RandomEquality re) {
  event done after(RandomEquality re)
            returning(boolean b) :
          call(* RandomEquality.hasNextNumber())
          && target(re) && condition(!b) {}
  event e0  after(RandomEquality re)
            returning(int i) :
          call(* RandomEquality.nextNumber())
          && target(re) && condition(i == 0) {}
  event e1  after(RandomEquality re)
            returning(int i) :
          call(* RandomEquality.nextNumber())
          && target(re) && condition(i == 1) {}
  event e2 after(RandomEquality re)
            returning(int i) :
          call(* RandomEquality.nextNumber())
          && target(re) && condition(i == 2) {}
  srs :
   e1 e0 -> e0 e1 . e2 e0 -> e0 e2 . e2 e1 -> e1 e2 .
   e0 e1 -> E        .
   E e1  -> e1 E     . E e0  -> e0 E     .
   E e2  -> #epsilon . e2 E  -> #epsilon .
   ^  done   -> #succeed .
   e0 done   -> #fail    .
   e1 done   -> #fail    .
   e2 done   -> #fail    .

  @succeed {
    System.out.println(
      p.toString() + " worked perfectly!"); }
  @fail {
    System.out.println(
      p.toString() + " failed!"); }
}
```

Figure 7.2: A JavaMOP specification that finds equality failures in the RandomEquality class

After the event definitions, we list the formalized property. The keyword srs tells JavaMOP that the following property will be a deterministic string rewriting system. Rules in our SRS formalism take the form "$l \rightarrow r$ .", meaning that the string of events on the left hand side of the arrow rewrites to that on the right side. The

three rules on the first line of this SRS sort the events: all e0 come before all e1 which come before all e2. The rule e0 e1 $\rightarrow$ E denotes that we have found a pair of e0 and e1, which must be matched by an e2.

Note that the SRS rules can be applied in any order when a new event is received, so it is user's responsibility to write *confluent* SRSs or to use the deterministic order of rule application explained in Section 7.2.1. The two rules on the third line move all instances of E to the right, so that they will eventually become adjacent to any instances of e2. The two rules on line four correspond to when such a situation occurs. When E is adjacent to e2 we have found a triple, and we can safely remove the symbol E from the string by rewriting it to #epsilon, which is a keyword specific to the SRS formalism in JavaMOP. The next rule is the success case, which occurs when done is at the beginning of the string (denoting that all 0's, 1's, and 2's have been equal). The symbol ^ corresponds to the beginning of the string. Similarly, $ corresponds to the end, but is not used here. #succeed is a special keyword that stops the rewriting process with the monitor signaling that a success was found. Like #succeed, #fail is a keyword that stops the monitor with a failure returned.

The next three lines are the failure cases; each rewrites to #fail. They occur when the number of 0's, 1's, and 2's was not equal, because the string will always be empty when a done occurs if they were properly balanced. The failure cases rely on the incremental nature of the string rewriting process. If taken as a normal SRS with complete strings as input, this would not be confluent. That means, the choice of what order to apply rules would result in different normal forms (See Section 7.2.1). Because the normal form is computed between the arrival of each event, e0, e1, or e2 can only occur before done in the string if an unequal number occurred. The string rewriting process is explained fully in Sections 7.2.2–7.2.4.

The last part of a JavaMOP specification is the handler section. Handlers are arbitrary Java code that is executed when the monitor raises a particular condition. Here the keywords @succeed and @fail denote that the code within the subsequent braces is run when the string rewrites to #succeed or #fail, respectively. In this example, the handlers simply print out informative messages when such situations occur. In general, handler code may be used for anything, such as running a specific algorithm or recovering from the error denoted by the failure of the safety property in question.

Aside from specifying properties which cannot be expressed by context-free grammars, string rewriting systems can be useful for expressing context-free and

finite properties in a natural, and often times more compact, form. Below are two properties from earlier papers [92, 93] written as SRSs.

The first property, called HASNEXT, is a property of the Java Iterator interface stating that hasNext() should always be called and return true before next is called. Below it is specified as a regular expression:

$$(\mathsf{hasnexttrue\ next})^*\ \mathsf{next}$$

The corresponding SRS is as follows:

$$\begin{aligned} \mathsf{hasnexttrue\ next} &\rightarrow \mathsf{\#epsilon} \\ \mathsf{hasnexttrue\ hasnexttrue} &\rightarrow \mathsf{hasnexttrue} \\ \mathsf{\hat{}next} &\rightarrow \mathsf{\#fail} \end{aligned}$$

While this SRS is certainly larger than the original ERE, it may be easier to understand by some users because it directly captures the semantics of the property by simply enumerating all the cases that one has to worry about. The rule hasnexttrue hasnexttrue $\rightarrow$ hasnexttrue conveys the notion that multiple calls to the hasNext() method are idempotent. hasnexttrue next rewrites to #epsilon because it is a safe operation. If next is seen at the beginning of the string a failure is raised as hasnexttrue was not properly called. Because our algorithm is incremental and deterministically rewrites from left to right it is not strictly necessary to match the beginning of the string, but it is more clear conceptually.

The second property is a properly context-free property called SAFELOCK which corresponds to the proper nesting of acquiring and releasing locks. Proper nesting, in this case, means that corresponding calls to acquire() and release() occur within the same method body. Here begin and end denote the beginning and end of a method body.

$$\begin{aligned} S &\rightarrow \epsilon \mid S\ \mathsf{acquire}\ M\ \mathsf{release}\ A \\ M &\rightarrow \epsilon \mid M\ \mathsf{begin}\ M\ \mathsf{end} \mid M\ \mathsf{acquire}\ M\ \mathsf{release} \\ A &\rightarrow \epsilon \mid A\ \mathsf{begin} \mid A\ \mathsf{end} \end{aligned}$$

The property is fairly complex, and a complete explanation can be found in [92].

The SRS for the property follows:

$$\begin{aligned} \text{begin end} &\rightarrow \#\text{epsilon} \\ \text{acquire release} &\rightarrow \#\text{epsilon} \\ \text{begin release} &\rightarrow \#\text{fail} \\ \text{acquire end} &\rightarrow \#\text{fail} \end{aligned}$$

In this case, the SRS is quite a bit less complex than the context-free grammar specifying the same safety property. Again, it conveys interesting semantic information. From the SRS it is clear that a begin followed immediately by a release() results in an error because we require all release() to occur in the same method call as the corresponding acquire(). Similarly, an acquire() follow by a end results in an error because the lock is not correctly released within the method body. begin end and acquire release rewrite to #epsilon because they are properly nested when they occur adjacently.

### 7.1.2 Chapter Contributions

There are two main contributions to this chapter:

- An efficient, optimized string rewriting algorithm. It builds upon a modification of the Aho-Corasick algorithm [3]. The original algorithm was designed for quickly finding strings in text. Our modified algorithm keeps track of substitution boundaries so that a rewrite step can be performed in time linear to the length of the right hand side of the matched rule.[4] To our knowledge, this is the first time it has been applied to string rewriting. An optimization has also been devised, which checks for early termination of rewriting.
- An implementation and extensive evaluation of the above algorithm as an MOP logic plugin for Runtime Verification. This way, it can serve as a specification formalism for parametric safety properties in instances of the MOP framework, such as JavaMOP. We show that its performance in practical Runtime Verification of large systems is acceptable when compared to other means to specify the same properties. Additionally, we show that it outperforms one of the state-of-the-art rewrite engines, Maude [39], which implicitly supports string rewriting as rewriting modulo associativity.

[4]The right hand side must be copied, so that the rule is still viable the next time it matches.

### 7.1.3 Chapter Outline

Section 7.2 presents our string rewriting algorithm, with its use and construction of pattern match automata and and optimization that allows for early termination. Section 7.3 presents our experimental results. Section 7.4 presents related work in the field of Runtime Verification, and Section 7.5 concludes.

## 7.2 Monitoring SRS Specifications

In this section, we present some basic notation for string rewriting systems and our string rewriting algorithm which was implemented as a logic plugin in the MOP framework.

### 7.2.1 Preliminaries

We refer the reader to [28] for an in-depth presentation of string rewrite systems. For an alphabet $\Sigma$, a *string rewriting system* (SRS) is a binary relation, $R$, on $\Sigma$, that is, a subset of $\Sigma^* \times \Sigma^*$. The set $\{\mathsf{l} \in \Sigma^* \mid (\mathsf{l}, \mathsf{r}) \in R\}$ is called the *domain* of $R$, denoted *dom*$(R)$, while similarly the set $\{\mathsf{r} \in \Sigma^* \mid (\mathsf{l}, \mathsf{r}) \in R\}$ is called the *range*, denoted *range*$(R)$. We refer here to any element $(\mathsf{l}, \mathsf{r}) \in R$ as a *rule* in $R$, any $\mathsf{l} \in$ *dom*$(R)$ as a *left hand side (LHS)* of a rule in $R$, and any $\mathsf{r} \in$ *range*$(R)$ as a *right hand side (RHS)* of a rule in $R$. In our SRS specifications in this chapter and in JavaMOP, rules $(\mathsf{l}, \mathsf{r}) \in R$ are written using the earlier shown syntax "$\mathsf{l} \rightarrow \mathsf{r}$".

The *single-step reduction relation* on $\Sigma^*$ that is induced by $R$ is defined as: for any $u, v \in \Sigma^*$, $u \rightarrow_R v$ if and only if there exists $(\mathsf{l}, \mathsf{r}) \in R$ such that for some $x, y \in \Sigma^*$, $u = x\ \mathsf{l}\ y$ and $v = x\ \mathsf{r}\ y$. The *reduction relation* on $\Sigma^*$ induced by $R$ is the reflexive, transitive closure of $\rightarrow_R$ and is denoted by $\rightarrow_R^*$. If for $x, y \in \Sigma^*$, $x \rightarrow_R^* y$ and $y$ is irreducible, $y$ is a *normal form* for $x$. $R$ is *confluent* if there is only one such $y$ for any given $x$, regardless of the order in which rules are applied.

In our SRSs in MOP, the symbols $s \in \Sigma$ correspond to either *events* of our property or symbols that appear in the RHS of rules in $R$. We call our string rewriting systems *deterministic* because the same normal form will always be chosen in the presence of a non-confluent $R$. Specifically, rules are applied left-to-right, with the smallest rule matching first in the case of overlap (e.g., for LHSs a a and a a b, the rule with a a as its LHS will always be applied first, starving the other rule). In the case of a conflict that is not resolved by the above, the order of

rules in the SRS specification is used to determine which rule to apply (e.g., if two rules have the same LHS, the one specified first will always be applied).

### 7.2.2 String Rewriting Algorithm Overview

There are two major parts to our SRS algorithm:

1. Finding matches of the LHSs of rules; and
2. Performing replacements with RHSs of rules.

To make replacements as efficiently as possible, the string of events/symbols that we rewrite is a linked listed of the SpliceList class, which was specially created for our purposes to allow constant time replacement of a section of the list with another list (splicing). The SpliceList class has a special type of Iterator defined for it, called the SLIterator, that does *not* follow the normal Iterator interface in Java.

Rather than only having next() and hasNext() methods, the SLIterator has next(int i), which moves the SLIterator forward i times and returns true if it is successful (i.e., does not reach the end of the SpliceList), and get(), which returns the current element that the SLIterator points to. SLIterator also has a method, splice(SLIterator second, SpliceList replacement), which takes another SLIterator to the same SpliceList and replaces the sequence denoted by those two SLIterators, inclusively, by a specified sequence replacement. It is because of the inclusive nature of the splice method that the SLIterator must have a method to retrieve its current element without advancing. The splice method makes it imperative for our string matching algorithm to maintain SLIterators to the beginning and end of the current LHS under consideration.

In Section 7.2.3 we discuss how this matching occurs using a modification of the Aho-Corasick string searching algorithm [3] that, unlike the base algorithm, keeps track of the beginning of a match, so that rewrites can be performed in constant time (after copying the RHS in time proportional to its length). To make this chapter self-contained, we give all the necessary information regarding the Aho-Corasick algorithm, rather than only this modification, but the modification is clearly delineated. To our knowledge, this is the first time any variation on the Aho-Corasick algorithm has been used in string rewriting, and no implementations of SRSs exist, that we could find. In Section 7.2.4, we present an in-depth explanation of how the pattern matching fits into the string rewriting algorithm

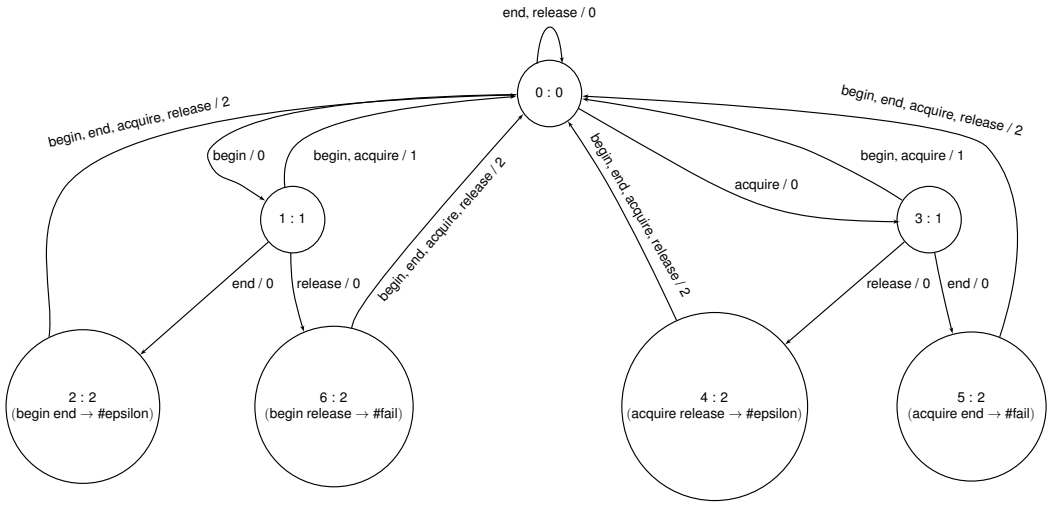



Figure 7.3: Pattern match automaton for the SAFELOCK property (see Section 7.1.1)

and how we optimize string rewriting to avoid considering sequences that cannot match any LHS.

### 7.2.3 Pattern Match Automata

The pattern match automata used by our string rewriting process, as mentioned, is a modification of the Aho-Corasick algorithm for finding strings in text [3]. The Aho-Corasick algorithm, which was originally not designed for string rewriting, is able to find all matches in a string in one linear pass, rather than performing separate passes for each rule LHS as would a naive matching algorithm. Our modification of the algorithm allows us to correctly adjust the SLIterator to the beginning of our current match, facilitating quick rewrites.

#### Using Pattern Match Automata

Fig. 7.3 shows the pattern match automaton for the SAFELOCK property. Each node has at least its state number and state depth, listed as a pair *number*:*depth*. The depth is used in two places in the automata generation algorithm, and simply states how many symbols (events) have been processed since the start state in one of the LHSs of the rewrite rules in our SRS. This will be explained in more detail below. Additionally, states which correspond to matching the left hand side of a given rule also display that rule, e.g., in state 6, the begin release → #fail rule is matched.

Each edge is marked by the list of symbols that cause that transition, as well as a number following a "/". That number, which we refer to as the *action*, is the number of times to increment the *first* SLIterator except in the self-transitions of state 0. When a self-transition in state 0 occurs, the *first* Iterator must be incremented once. When a forward transition with "/ 0" is encountered, a transition to the next state is made, and the next input is considered. If the transition is suffixed with something *other* than 0, the transition must be a backward transition, and the same symbol that is currently under consideration must be evaluated in the next state. This is why we handle self-transitions in state 0 as a special case, if it were suffixed with "/ 1" and handled as a backward transition, the same symbol would be considered infinitely.

Fig. 7.4 shows the pseudocode for pattern matching using a given pattern match automaton. The only global variable for the algorithm is the given PatternMatchAutomaton, *pma*. The algorithm begins by initializing the *first* and *second* SLIterators to the beginning of the argument SpliceList $l$, using the *head()* method. The local *currentState* is initialized to the initial machine state, here represented as 0.[5] The while loop beginning on line 10 will only exit when the end of $l$ is reached, denoted by the break statements on lines 20 and 25. We know that the end of $l$ is reached on lines 20 and 25 when the next(int i) method returns false. We never need to check if *first.next* returns false because it may never advance past *second* due to the construction of the PatternMatchAutomaton. Lines 17–22 cover the self transition to state 0 mentioned earlier, while lines 23–27 represent a normal forward transition. 23–27 are a forward transition because the *action* of the transition is 0. As mentioned earlier, the only difference between the 0 self-transition and a forward transition is that in the self-transition the *first* SLIterator need be incremented (line 18). Lines 28–30 handle a backward transition in the PatternMatchAutomaton. As expected, with a backward transition the *first* SLIterator is incremented a number of times specified by the *action* of *transition* and *second* is *not* incremented so that the same symbol will be considered in the next iteration of the loop. One interesting property of this algorithm is that if one pattern is a prefix of another, such as the patterns "$a\ a\ \rightarrow c$" and "$a\ a\ b\ \rightarrow d$", both matches will be reported. This is undesirable behavior for rewriting because "$aa$" will be rewritten to $c$ immediately and "$a\ a\ b$" should no longer be matchable. This will be accounted for in Section 7.2.4.

As an example of how the pattern match algorithm functions, suppose that the

[5]It is actually a class that may contain a matched rule, as we can see in Fig. 7.3.

```
1   globals PatternMatchAutomaton pma
2   locals SLIterator first, second
3          State currentState, nextState
4          Symbol symbol
5          Transition transition
6   procedure match(SpliceList l)
7   first ← l.head()
8   second ← l.head()
9   currentState ← 0
10  while (true){
11     if (currentState.hasMatch()){
12        //signal match
13     }
14     symbol ← second.get()
15     transition ← pma.get(currentState, symbol)
16     nextState ← transition.state
17     if (nextState = 0){
18        first.next(1)
19        if (¬second.next(1)){
20           break
21        }
22     }
23     else if (transition.action = 0){
24        if (¬second.next(1)){
25           break
26        }
27     }
28     else {
29        first.next(transition.action)
30     }
31     currentState ← nextState
32  }
```

Figure 7.4: Pattern Match Algorithm

| current state | symbol | next state | *first* index |
|---|---|---|---|
| 0 | begin | 1 | 0 |
| 1 | begin | 0 | 1 |
| 0 | begin | 1 | 1 |
| 1 | acquire | 0 | 2 |
| 0 | acquire | 3 | 2 |
| 3 | begin | 0 | 3 |
| 0 | begin | 1 | 3 |
| 1 | end | 2 | 3 |

Figure 7.5: A run of the pattern match algorithm on begin begin acquire begin end

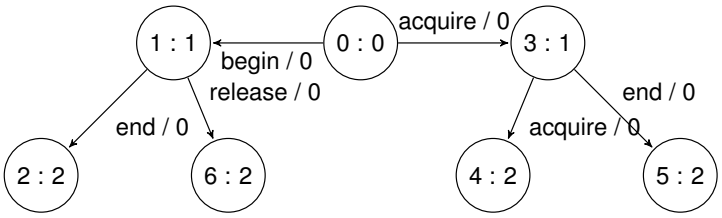



Figure 7.6: Forward Transitions for SAFELOCK (matched rules omitted)

following series of events have been seen at a given point in a program: begin begin acquire begin end. At this point, the SAFELOCK property will experience its first match of a rule LHS. Fig. 7.5 shows the state transitions as each symbol is considered, as well as the position of the *first* SLIterator. An important thing to note is that every time we transition back to state 0, the *first* SLIterator index is incremented by 1 (specified by the back transitions), and the symbol is evaluated again in state 0. In general, back transitions need not be to state 0, as we shall see. At the end of the input, the algorithm is in state 2, which matches the rule begin end $\rightarrow$ #epsilon. The *first* SLIterator correctly points to index 3, which is the last begin event. The *second* SLIterator always points at the current input, which is end. These SLIterators can then be used to quickly replace begin end with #epsilon, as we will see in Section 7.2.4.

### Generating Pattern Match Automata

There are two main phases to the creation of pattern match automata. In the first phase the forward transitions of the automaton are created. In the second phase,

all of the backward transitions and the self-transition that (almost) always exists in state 0 are added. During the computation of the backward transitions, the actions for the backward transition are also computed and added to the backward transitions. As mentioned, only backward transitions ever have non-0 actions, since they correspond to places in the automaton where there is a switch from matching one potential set of LHSs of rules to another. For instance, in Fig. 7.5, between the third begin and the first acquire, there is a switch from potentially matching {begin end, begin release} to {acquire end, acquire release}, which requires no longer considering the begin event for match purposes, thus the action of 1.

To create the forward transitions for an automaton, we add one path that corresponds directly to the left hand side of each rule in our string rewriting system. We add these paths one at a time, and reuse as many states as possible. Each forward transition is assigned the action 0. Fig. 7.6 shows the forward transitions for the pattern match automaton originally presented in Fig. 7.3. For each LHS, we begin at state 0 and add a transition for the first symbol. Because all patterns SAFELOCK begin with either begin or acquire, we have only two transitions, one labeled with begin and one labeled with acquire. We continue to transitively add transitions based on the remainder of each LHS. For the two rule LHSs beginning with begin, one ends with end and the other ends with release, so there are two transitions out of state 1 labeled accordingly. As each new state is added to the machine during the forward transition phase, the depth of the state is recorded. The depth is simply the number of symbols from state 0. For instance, state 6 is at depth 2, since two symbols, begin followed by end, lead to state 6. The largest depth always corresponds to the longest rule LHS.

In the second phase, the self-transition on state 0 is added first, if needed. The self-transition is only necessary if there is not a forward transition out of state 0 for every symbol used in the SRS or specified by the JavaMOP front end.[6]

After potentially adding the self-transition in state 0, the backward transitions are added to the pattern match automaton. Backward transitions are only added from a given state for symbols that do not have forward transitions out of that state. All backward transitions from a given state, $s$, will go to the same place, so we define $\mathit{fail}(s) = s'$, where $s'$ is the destination of a backward transition out of $s$. To find the destination for the backward transitions out of a state in pattern match automaton *pma* with depth $d$, we consider each state $r$ of depth $d - 1$ and perform

[6]JavaMOP allows one to define events that do not appear in the specified property; these will correspond to symbols that are never rewritten by the specified SRS.

the following actions, transitions are added in depth first order [3]:

1. If *pma.get*$(r, a)$ is a backward transition for all symbols $a$, do nothing.
2. Otherwise, for each symbol $a$ such that *pma.get*$(r, a)$ = $s$, do the following:
   (a) Let $s' = \mathit{fail}(r)$.
   (b) Compute $s' \leftarrow \mathit{fail}(s')$ until such point as *pma.get*$(s', a)$.*action* $= 0$. Because state 0 must have either a forward transition or a self-transition for every symbol, such an $s'$ must exist.
   (c) For all $a'$ such that *pma.get*$(s, a')$ has no forward transition, assign *pma.get*$(s, a')$.*state* $= s'$, ***pma.get*$(s,a')$.*action* = *s.depth* - *s'.depth***.

The procedure above is essentially the same as [3]. The part in bold is specific to our algorithm for string rewriting. The action is assigned as such because the depth of a given state represents the number of symbols processed since state 0 in the automaton, thus the difference in the depths tell us the number of symbols that we need to skip with the *first* SLIterator in Fig. 7.4. While the pattern match automaton for SAFELOCK has backward transitions that only go to state 0, as mentioned, this is not always the case in general. When the suffix of one LHS overlaps with the prefix of another, backward transitions that do not go back to state 0 are generated. An example of this can be seen in Fig. 7.7, where the SRS in question is b a a $\rightarrow$ #epsilon, a a c $\rightarrow$ #epsilon. Because b a a and a a c have a suffix/prefix overlap, the backward transitions from state 3 at depth 3 go to state 5 at depth 2, resulting in an action of only 1. For example, consider input b a a c. When we switch from matching b a a to matching a a c, which occurs between states 3 and 5, we wish to only "forget" the b at the beginning, an action of 1.

### 7.2.4 Rewriting using Pattern Match Automata

The rewriting algorithm we use to monitor SRS's is presented in Fig. 7.8. Not pictured in Fig. 7.8, is the action of the monitor itself. As any monitoring algorithm in the MOP framework, events arrive one at a time. As each event occurs, we add it—as a symbol representing that event—to a SpliceList that contains the results of rewriting previous sequences of events. Additionally, if any rules make use of the ^ symbol, it will be added to the beginning of the SpliceList and treated as a

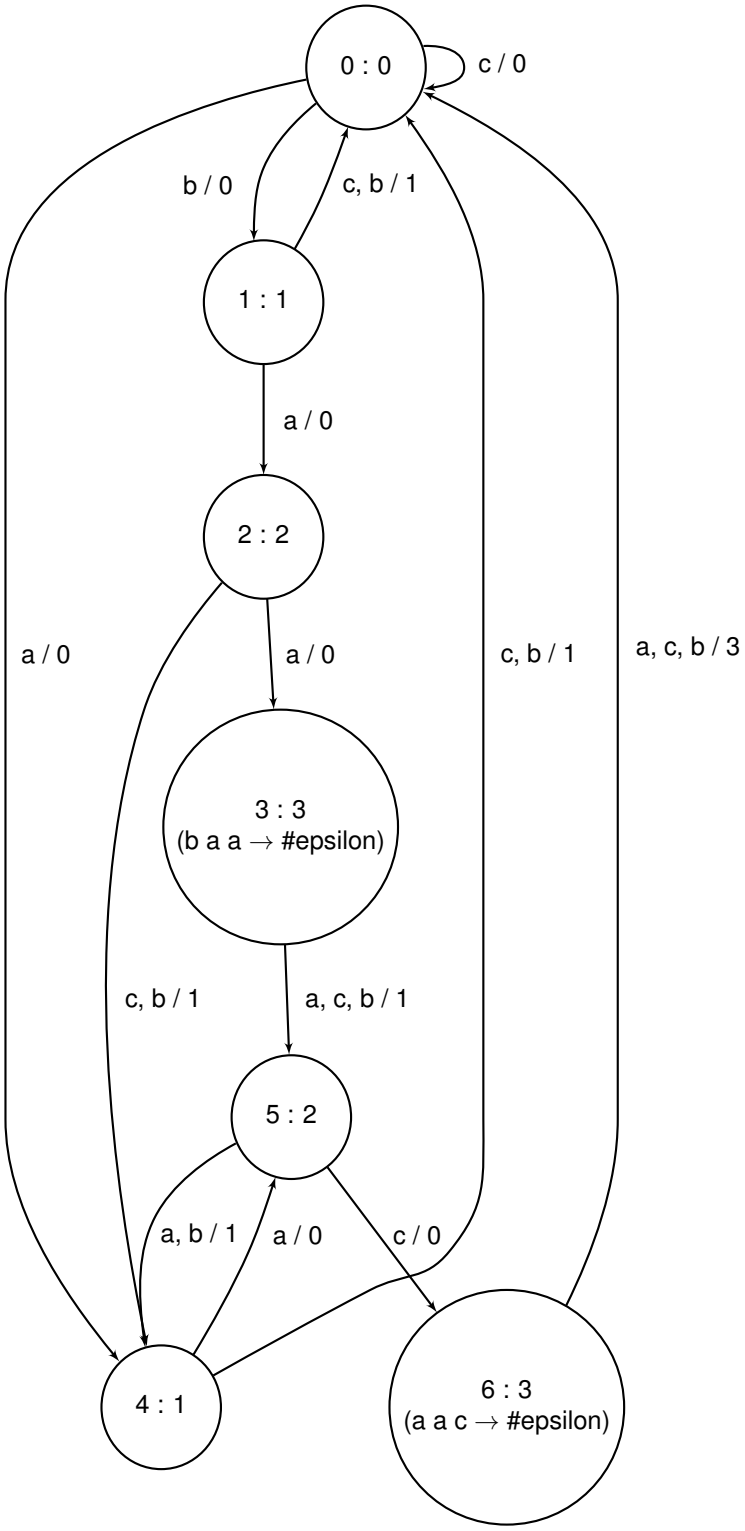
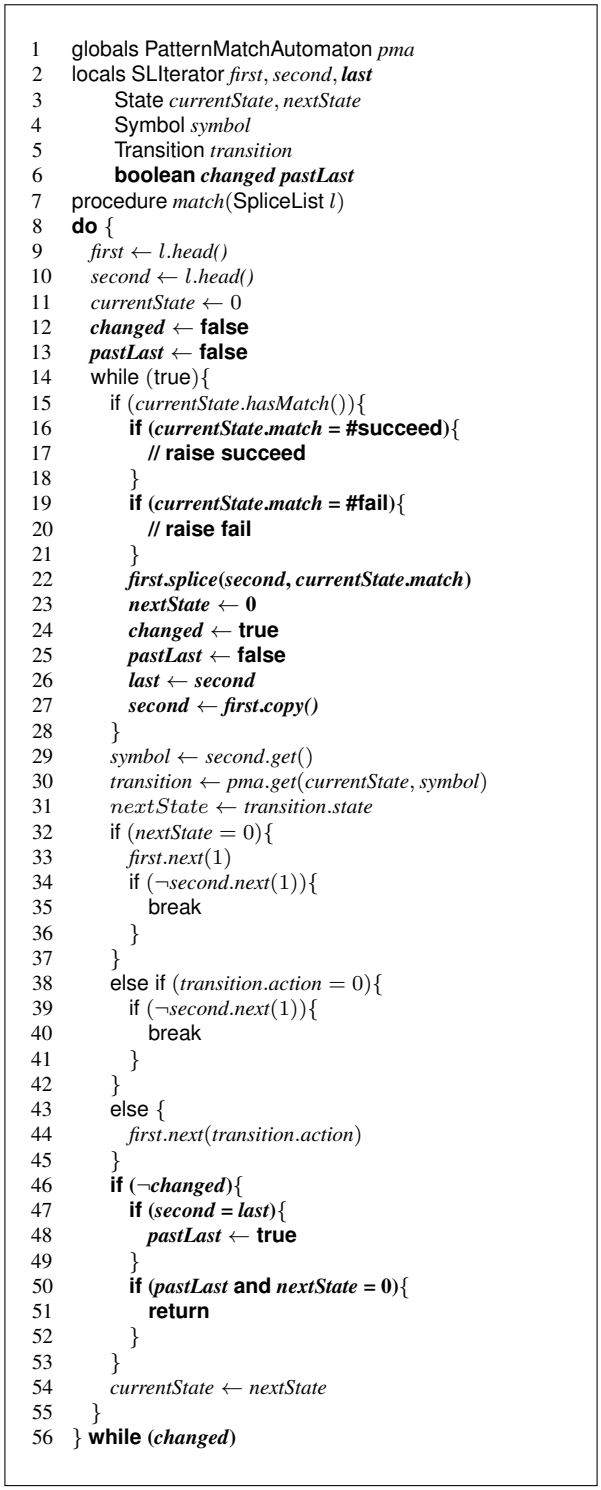



Figure 7.7: A pattern match automaton with overlap



```
1   globals PatternMatchAutomaton pma
2   locals SLIterator first, second, last
3          State currentState, nextState
4          Symbol symbol
5          Transition transition
6          boolean changed pastLast
7   procedure match(SpliceList l)
8   do {
9     first ← l.head()
10    second ← l.head()
11    currentState ← 0
12    changed ← false
13    pastLast ← false
14    while (true){
15      if (currentState.hasMatch()){
16        if (currentState.match = #succeed){
17          // raise succeed
18        }
19        if (currentState.match = #fail){
20          // raise fail
21        }
22        first.splice(second, currentState.match)
23        nextState ← 0
24        changed ← true
25        pastLast ← false
26        last ← second
27        second ← first.copy()
28      }
29      symbol ← second.get()
30      transition ← pma.get(currentState, symbol)
31      nextState ← transition.state
32      if (nextState = 0){
33        first.next(1)
34        if (¬second.next(1)){
35          break
36        }
37      }
38      else if (transition.action = 0){
39        if (¬second.next(1)){
40          break
41        }
42      }
43      else {
44        first.next(transition.action)
45      }
46      if (¬changed){
47        if (second = last){
48          pastLast ← true
49        }
50        if (pastLast and nextState = 0){
51          return
52        }
53      }
54      currentState ← nextState
55    }
56  } while (changed)
```

Figure 7.8: Rewriting Algorithm

normal symbol by the rewriting algorithm. As for uses of $, the current event must be added *before* $.[7]

After an event is added to the SpliceList, the algorithm in Fig. 7.8 is evaluated to completion before another event can be accepted. The algorithm is similar to the pattern match procedure of Fig. 7.4. The changes are in bold. There are three main changes: the inclusion of a loop that ensures that a normal form is reached, the actual rewriting step itself, and a section that recognizes early termination.

The first new control structure to notice is the do...while loop from line 8 to 56. This loop ensures that rewriting continues until there is a pass through the loop in which nothing changes, i.e., the string is in normal form. The new boolean variable, *changed*, controls this loop. It is set to false at the beginning of an iteration of the do...while loop, and to true on line 24, which is only executed when a rewrite occurs.

Lines 15–28 perform the actual rewriting step. The element *match* of a State contains the right hand side of the rule matched in that State. If the *match* is one of the two special keywords #succeed or #fail, a success or fail handler is executed, as appropriate, and rewriting terminates. If either handler is executed, the monitor is considered dead unless it is reset (see [93]). If *match* is something else, the splice method is called on line 22. The splice method is a special method of SLIterator that replaces a range specified by the this and an argument SLIterator with the argument sequence. Here the range is specified by *first* and *second*, and *currentState.match* is passed as the replacement. Note that if the right hand side of the rule is #epsilon, it is represented as an empty sequence, which splice is able to handle. The splice method also correctly sets *first* and *second* to point to the beginning and end of the spliced in *match* sequence, or the next symbol if *match* was #epsilon. On line 26, we set *last* to *second*, so that *last* points to the end of the last replacement, this will be used to determine early termination. Then, on line 27, *second* is set as a copy of *first*. This ensures that segments of string which are transitively rewritten will be rewritten immediately. Because splice changes the SpliceList, it is important to set *currentState* back to state 0 because any matching will occur in the newly rewritten segment of the SpliceList.

In the last new addition to the match algorithm, from lines 46 to 53, we test for early termination of the algorithm. The idea here is to exit early if we enter a segment of the SpliceList that we know for certain cannot be rewritten. This

[7] Because of this there is a very small performance hit for using $ in a rule, but ^ is essentially free.

| event | initial $l$ | $l$ in normal form |
|---|---|---|
| begin | begin | begin |
| end | begin end | #epsilon |
| begin | begin | begin |
| acquire | begin acquire | begin acquire |
| release | begin acquire release | begin |
| acquire | begin acquire | begin acquire |
| end | begin acquire end | #fail |

Figure 7.9: An SRS monitoring run for SAFELOCK

happens when we reach a point that is past the end of the *last* SLIterator, which was set in a previous iteration, no rewrites have occurred in the current iteration, and *currentState* returns to 0. The first two requirements are fairly straight-forward: if a change occurs, new matches are possible, and if we are in a segment of the SpliceList before the last rewrite, we are still investigating symbols that are potentially new. However, if there is no rewrite in the current iteration and we are past the last change from the previous iteration, we are seeing symbols that were seen in the previous iteration with no change. The last condition, that we must return to State 0, is more subtle. The reason for this is that there could have been a rewrite in the last iteration that inserted a segment that appears in the middle of a left hand side of one of the rules. A simpler way to look at this requirement is that if *pma* is not in state 0 it is actively matching *something*. This condition for early termination can lead to an unbounded amount of saving, as the SpliceList can be of an unbounded length.

Fig. 7.9 shows a monitor run as non-parametric events for SAFELOCK arrive. The non-parametric events are dispatched to the correct monitor instance by the indexing of JavaMOP (or whatever projection method is used in future language instances of MOP). The first column shows the arriving event, the second column shows the state of the SpliceList $l$ before any rewriting, and the last column shows the normal form for $l$ after the rewriting algorithm of Fig. 7.8 has run. After the last event a failure has occurred, and the fail handler will execute.

| Benchmark | Original (ms) | HASNEXT | | SAFESYNCCOL | | SAFESYNCMAP | | UNSAFEITER | | UNSAFEMAPITER | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | ERE | SRS | ERE | SRS | ERE | SRS | ERE | SRS | ERE | SRS |
| avrora | 2317* | 194 | 227 | 35* | 103* | 28 | 120* | 253* | 288* | 41 | 134* |
| batik | 773 | 0 | 6 | 5 | 11 | 9 | -1 | 5 | 3 | 1 | 2 |
| eclipse | 11749 | -1 | -2 | -2 | -4 | -1 | -3 | -2 | -2 | -2 | -2 |
| fop | 251 | 922 | 2091 | 26 | 24 | 21 | 20 | 34 | 57 | 28 | 42 |
| h2 | 3860 | 9 | 15 | 6 | 2 | 0 | 4 | 15 | 22 | 8 | 24 |
| jython | 1400 | 3 | 4 | 3 | 3 | 4 | 2 | 16 | 18 | 3 | 3 |
| luindex | 478 | 2 | -1 | 2 | 0 | 0 | 4 | 1 | 2 | 0 | -5 |
| lusearch | 581 | 1 | 3 | -1 | 1 | 3 | 3 | 46 | 46 | 2 | 0 |
| pmd | 1441 | 27 | 117 | 139 | 137 | 10 | 17 | 72 | 148 | 177 | 199 |
| sunflow | 1222 | 5 | 8 | 0 | -1 | 6 | -3 | -4 | 4 | 0 | 3 |
| tomcat | 1068 | 2 | 4 | 3 | 3 | 3 | 1 | 2 | 2 | 2 | 2 |
| tradebeans | 4618 | 2 | 1 | -1 | -3 | -1 | 2 | 4 | -2 | -2 | -1 |
| tradesoap | 3213 | 1 | -1 | 1 | -1 | 0 | -2 | 1 | 0 | 0 | 0 |
| xalan | 359 | 5 | 1 | 5 | 1 | 6 | 3 | 90 | 172 | 7 | 8 |

Figure 7.10: Comparison of JavaMOP with extended regular expressions (ERE) and with the same properties expressed as string rewriting systems (SRS): average *percent* overhead (convergence within 3% except those marked with *)

## 7.3 Evaluation

Our SRS implementation is evaluated in two contexts: first we show how it compares, within the context of JavaMOP, to finite-state logics on the DaCapo benchmark suite [22]. Then we give a comparison of our underlying SRS rewrite engine against the Maude [39] term rewriting engine, modulo associativity. The goal of the first evaluation is to show that SRS monitoring is efficient enough to be used in large programs, being not much less efficient than finite-state logics (extended regular expressions in this case). The goal of the second experiment is to show that our SRS implementation is more efficient than the state-of-the-art.[8]

All experiments were performed on a machine with a 3.82GHz Intel® Core™ i7 970 hexcore with Hyper-Threading (12 hardware threads) and 24 GB of ram. Ubuntu 11.10 64 bit was used as the operating system and version 9.12 of DaCapo was used as the benchmark suite, with default inputs and the -converge option to gain convergence within 3%. OpenJDK version 1.6.0_23 as the Java virtual machine. All compiled JavaMOP specs were weaved into DaCapo using ajc 1.6.11. Maude 2.6 was used for comparison with Maude.

The following properties were used in the DaCapo experiments. The SRS versions of them (shown below) are new, while the extended regular expression versions were borrowed from [24, 26, 31, 92].

[8]Note that Maude is more general than our SRS engine, but there is a price for that generality, and general term rewriting makes little sense in the context of MOP event traces.

- HASNEXT: Do not use the next element in an Iterator without checking for the existence of it (see Section 7.1.1);
- SAFESYNCCOL: If a Collection is synchronized, then its iterator also should be accessed synchronously:

$$\begin{aligned} \text{sync asyncCreateIter} &\rightarrow \#\text{fail} \\ \text{sync syncCreateIter accessIter} &\rightarrow \#\text{fail} \end{aligned}$$

- SAFESYNCMAP: If a Collection is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner:

$$\begin{aligned} \text{sync createSet asyncCreateIter} &\rightarrow \#\text{fail} \\ \text{sync createSet syncCreateIter accessIter} &\rightarrow \#\text{fail} \end{aligned}$$

- UNSAFEITER: Do not update a Collection when using the Iterator interface to iterate its elements:

$$\begin{aligned} \text{update use} &\rightarrow \#\text{fail} \\ \text{use use} &\rightarrow \text{use} \\ \text{update update} &\rightarrow \text{update} \\ \text{createIterator} &\rightarrow \#\text{epsilon} \end{aligned}$$

- UNSAFEMAPITER: Do not update a Map when using the Iterator interface to iterate its values or its keys:

$$\begin{aligned} \text{update use} &\rightarrow \#\text{fail} \\ \text{use use} &\rightarrow \text{use} \\ \text{update update} &\rightarrow \text{update} \\ \text{createIterator} &\rightarrow \#\text{epsilon} \\ \text{createCollection} &\rightarrow \#\text{epsilon} \end{aligned}$$

For the comparison with Maude, strings of equal numbers of 2's, 1's, and 0's, with the 2's preceding the 1's preceding the 0's were generated, and the following rewrite system applied. Note, that the language of strings that reduce to #epsilon with this rewrite system is non-context free. It is very similar to EQUALITYCHECK from Section 7.1.1.

| N | Maude Time (ms) | SRS Time (ms) |
|---|---|---|
| 100 | 42 | 33 |
| 1000 | 37038 | 236 |
| 5000 | DNF | 7112 |
| 10000 | DNF | 26132 |

Figure 7.11: Comparison of maude versus SRS rewrite. DNF: did not finish in one hour

| | | | | | |
|---|---|---|---|---|---|
| 1 0 | → | 0 1 | 2 0 | → | 0 2 |
| 2 1 | → | 1 2 | 0 1 | → | 3 |
| 1 3 | → | 3 1 | 3 0 | → | 0 3 |
| 3 2 | → | #epsilon | 2 3 | → | #epsilon |

Fig. 7.10 shows a comparison of finite-state properties specified in JavaMOP using ERE and SRS. The first column shows the individual DaCapo [22] benchmarks, and the second column shows runtime of the original uninstrumented benchmarks in milliseconds. All other columns are *percent* overhead. Each benchmark-property pair converged to within 3% except the instances of avrora marked with *. The results presented for avrora that did not converge are the average of twenty runs with outliers removed, but they are still not as trustworthy as the converging results. This lack of convergence is a problem on highly multithreaded machines. We can see that even the uninstrumented, original run, fails to converge. Negative overheads are the result of noise in the experimental settings and changes in code layout due to instrumentation resulting in slightly more efficient programs.

Overall, the average overhead on the DaCapo benchmark suite was 58% for SRS, while it was 33% for ERE. When fop-HASNEXT—which has, by far, the worst overhead of any trial—is removed from both, the overhead drops to 29% and 20%, respectively. It must be noted, that the properties we use are specifically selected for generating large overheads; they are very intensive properties that generate *many events* (see [76]). The overhead numbers are slightly larger than reported in previous papers because we have moved to a multi-threaded, and quite simply faster, machine. The monitors in JavaMOP must be synchronized, which results in higher overhead for programs that actually make use of multiple threads. Any monitoring system must do the same thing if the monitors are for cross-thread properties (like all of those properties used here). In most of the

benchmark/property pairs, the performance of ERE and SRS are very comparable. For pmd-HASNEXT and avrora-SAFESYNCMAP, SRS shows more than three times the overhead of ERE, but for all other trials SRS is never more than three times worse.

Fig. 7.11 shows the comparison of Maude to our SRS engine with the rewrite system discussed above. N refers to the number of each digit, i.e., N=100 has 300 characters in it: 100 each of 2, 1, and 0. As we can see from the results, our SRS engine runs in 78% of the time of maude at N=100. At N=1000, our SRS engine runs in .006% of the time of Maude. With larger inputs, Maude fails to complete in an hour, while our SRS engine takes less than 30 seconds on every tested input.

## 7.4 Chapter Related Work

Many approaches have been proposed to monitor program execution against formally specified properties (see the summary of related work in the Introduction to this thesis). Briefly, all runtime monitoring approaches except MOP have their specification formalisms hardwired, and few of them share the same logic.

There are four orthogonal attributes of a runtime monitoring system: logic, scope, running mode, and handlers. The logic specifies which formalism is used to specify the property. The scope determines where to check the property; it can be class invariant, global, interface, etc. The running mode denotes where the monitoring code runs; it can be inline (weaved into the code), online (operating at the same time as the program), outline (receiving events from the program remotely, e.g., over a socket), or offline (checking logged event traces). The handlers specify what actions to perform under exceptional conditions; such conditions include violation and/or validation of the property. It is worth noting that for some logics, violation and validation are not complementary to each other, i.e., the violation of a formula does not always imply the validation of the negation of the formula. MOP allows for handlers for any number of user defined exceptional situations (called handler categories).

Most runtime monitoring approaches can be framed in terms of these attributes, as illustrated in Fig. 7.12, which shows an (incomplete) summary of runtime monitoring systems. For example, JPaX can be regarded as an approach that uses linear temporal logic (LTL) to specify class-scoped properties, whose monitors work in offline mode and only detect violation. In general, JavaMOP (the Java

| Approach | Logic | Scope | Mode | Handler |
|---|---|---|---|---|
| JPaX [59] | LTL | class | offline | violation |
| TemporalRover [44] | MiTL | class | inline | violation |
| JavaMaC [82] | PastLTL | class | outline | violation |
| Hawk [41] | Eagle | global | inline | violation |
| RuleR [17] | RuleR | global | inline | violation |
| Tracematches [10] | Reg. Exp. | global | inline | validation |
| J-Lo [23] | LTL | global | inline | violation |
| Pal [29] | modified Blast | global | inline | validation |
| PQL [89] | PQL | global | inline | validation |
| PTQL [54] | SQL | global | outline | validation |

Figure 7.12: A Selection of Monitoring Systems

instance of MOP) has proven to be the most efficient of the runtime monitoring systems despite being generic in logical formalism.

Of the systems mentioned in Fig. 7.12, only PQL [89], Hawk/Eagle [41], and RuleR [17] provide logical formalisms with greater than finite-state power. Hawk/Eagle adopts a Turing-complete fix-point logic, but it has problems with large programs because it does not garbage collect the objects used in monitoring. In addition, Hawk/Eagle is not publicly available.[9] Because of this and the fact that Hawk/Eagle has not been run on DaCapo [22] with the same properties, we cannot compare JavaMOP with our new string rewriting systems plugin with Hawk/Eagle. RulerR is a rule-based monitoring system which has the ability to also specify Turing complete properties. The current implementation of RuleR is not built for efficiency, and is, additionally, not publicly available. PQL is not Turing-complete, and performance comparisons with PQL using an older, less efficient, version of JavaMOP can be found in [92]. String rewriting was used in the context of monitoring for detection of malware in [19]. This was, in many ways, the inspiration for adding string rewriting to MOP. However, the string rewriting patterns allowed in that work were regular (i.e., can capture only regular languages), while our goal is to provide a true Turing-complete logical formalism for parametric monitoring.

MOP [32, 33] is an extensible Runtime Verification framework that provides efficient, logic-independent support for parametric specifications. JavaMOP is an instance of MOP for the Java programming language. It allows the developer to

[9] [10] makes an argument for the inefficiency of Hawk/Eagle. Since Hawk/Eagle is not publicly available (only its rewrite based algorithm is public [41]), the authors of Hawk/Eagle kindly agreed to monitor some of the simple properties from [26]. We have confirmed the inefficiency claims of [10] with the authors of Hawk/Eagle.

specify desired properties using formal specification languages, along with code to execute when properties are matched or fail to match. Monitoring code is then automatically generated from the specified properties and integrated together with the user-provided code into the original system.

MOP is a highly extensible and configurable Runtime Verification framework. The user is allowed to extend the MOP framework with his/her own logics via *logic plugins* which encapsulate the monitor synthesis algorithms. This extensibility of MOP is supported by an especially designed layered architecture [32], which separates monitor generation and monitor integration. By standardizing the protocols between layers, modules can be added and reused easily and independently. MOP also provides efficient and logic-independent support for *parametric* parameters [31], which is useful for specifying properties related to groups of objects. This extension allows associating parameters with MOP specifications and generating efficient monitoring code from parametric specifications with monitor synthesis algorithms for non-parametric specifications. MOP's generic support for parametric patterns simplified our SRS plugin's implementation.

The JavaMOP instance provides two interfaces: a web-based interface and a command-line interface, providing the developer with different means to manage and process JavaMOP specifications. AspectJ [79] is employed for monitor integration: JavaMOP translates outputs of logic plugins into AspectJ code, which is then merged within the original program by an AspectJ compiler. Seven logic-plugins are currently provided with JavaMOP: finite state machines, extended regular expressions, context-free grammars, past time linear temporal logic, linear temporal logic with past and future operators, past time linear temporal logic with calls and returns, and, now, string rewriting systems. Descriptions of the first six plugin-ins can be found in [93].

## 7.5 Chapter Conclusion

We provided the first means to efficiently monitor parametric Turing-complete specifications using string rewriting systems. By using a modified version of the Aho-Corasick string matching algorithm and a means to terminate the rewriting process early, the resultant string rewriting algorithm is quite practical, as shown in our extensive evaluation.[10] The average overhead on the DaCapo benchmark suite

[10]Special thanks to Dongyun Jin for help with DaCapo experimental settings.

was 58% for SRS, while it was 33% for ERE. When the largest benchmark/property pair is removed from both, the overhead drops to 29% and 20%, respectively. A less extensive comparison of our core string rewriting algorithm with the term rewrite engine Maude, which provides implicit support for string rewriting through its rewriting modulo associativity, suggests that our approach can lead to new string rewriting engines that outperform the state-of-the-art.

# Chapter 8

# Predictive Analysis

## 8.1 Chapter Introduction

This chapter presents work on the RV-Predict system, which is a thorough re-engineering of the earlier jPredictor system [37]. While it is a complete system, there is still room for improvements in efficiency. The RV-Predict system has been an effort of the company Runtime Verification, Inc., co-founded in 2010 by Grigore Roşu and this author.[1]

The current trend in processor design is forcing an ever increasing importance on the proper design of concurrent software systems to maintain the improved performance trends users expect as a result of Moore's law. Concurrent systems in general and multithreaded systems in particular may exhibit different behaviors when executed at different times. This inherent non-determinism makes multithreaded programs difficult to analyze, test and debug. Predictive analysis is able to detect, correctly, concurrency errors from observing execution traces of multithreaded programs. By "correct" or "sound" prediction of errors we mean that there are *no false alarms*. The program is automatically instrumented to emit runtime events, and a causal model built from the observed traces. The particular execution that is observed need *not* hit the error; yet, errors in other executions can be correctly predicted together with counter-examples leading to them.

There are several other approaches also aiming at detecting potential concurrency errors by examining particular execution traces. Some of these approaches aim at verifying general purpose properties [110, 112], including temporal ones, and are inspired from debugging distributed systems based on Lamport's *happens-before* causality [84]. Other approaches work with particular properties, such as data-races and/or atomicity. [108] introduces a first lock-set based algorithm to

[1] All work in this chapter is an extension of the earlier work on jPredictor by Traian Şerbănuţă, Feng Chen, and Grigore Roşu. The work here was performed in association with Dennis Griffith, Michael Ilseman, and Grigore Roşu.

detect data-races dynamically, followed by many variants aiming at improving its accuracy. For example, an ownership model was used in [124] to achieve a more precise race detection at the object level. [97] combines the lock-set and the happen-before techniques. The lock-set technique has also been used to detect atomicity violations at runtime, e.g., the reduction based algorithms in [51] and [125]. [125] also proposes a block-based algorithm for dynamic checking of atomicity built on a simplified happen-before relation, as well as a graph-based algorithm to improve the efficiency and precision of runtime atomicity analysis.

Previous efforts tend to focus on either soundness or coverage: those based on happens-before try to be sound, but have limited coverage over interleavings, thus missing errors; lock-set based approaches have better coverage but suffer from false alarms. RV-Predict aims at improving coverage without giving up soundness or genericity of properties. It combines *sliced causality* [34], a happen-before causality drastically—but soundly—sliced by removing irrelevant causalities using semantic information about the program obtained with an apriori static analysis, with *lock-atomicity*. Our predictive runtime analysis technique can be understood as a hybrid of testing and model checking. Testing because one runs the system and observes its runtime behavior in order to detect errors, and model checking because the special causality with lock-atomicity extracted from the running program can be regarded as an abstract model of the program, which can further be investigated exhaustively by the observer in order to detect potential errors.

### 8.1.1 Chapter Contributions

This chapter primarily showcases engineering improvements to the jPredictor system that make it efficient and effective.[2] Before these engineering changes, jPredictor simply did not work for realistic programs. jPredictor had no memory buffering for traces read from disk, and multiple trace reversals were necessary. Traces were completely uncompressed, and lastly, working sets aside from the traces were contained within memory only. This last fact causes jPredictor literally to crash for significant programs, as it runs out of memory. RV-Predict backs all working sets with a hashing database; it can never run out of memory, only disk space. Additionally, it uses a pipelined approach with memory buffering that results in only accessing each event on disk once, whereas jPredictor performed each stage

[2]The only remaining code from jPredictor is the calculation of termination-sensitive control dependence, however.

in serial fashion. RV-Predict still is not perfect, however. While it is multiple orders of magnitude more efficient than jPredictor, and can work on examples for which jPredictor does not, it still has problems with very large programs. Obviously, running out of disk space is always an issue. It also runs into heavy garbage collection churn for traces with particularly large causal model spaces. Ultimately, we still feel that RV-Predict is a large step in the right direction.

Two theoretical contributions are also presented: by loop peeling instrumentation we are able to find the same races as jPredictor, while vastly reducing the size of traces, and we also present the first algorithm that is actually capable of traversing the search space induced by the causal model for generic properties specified using JavaMOP. While [37] makes the theoretical claim to supporting generic property checking, no algorithm is given, and in fact, jPredictor had no implementation. Our algorithm induces the minimum number of interleavings possible to correctly predict violations of properties. Incidentally, a specialized version of this algorithm is included for race detection that, while still having the theoretical $n^2$ upper bound of comparisons required by the jPredictor race detector, in practice, has several orders of magnitude fewer comparisons, as it avoids examining causally impossible pairs of reads/writes.

## 8.2 RV-Predict Overview

To understand RV-Predict, it is instructive to begin with a high level overview of the system, so that the various necessities of the system may be understood. Fig. 8.1 shows the pipeline for the RV-Predict system. The stages in stages in Fig. 8.1 can be broken down into three phases, described in the next three sections: generating event logs (Section 8.3, building a causal model (Section 8.4), and property checking (Section 8.5). The first two phases consist of two stages each, the last phase has only one stage. While the first two stages of the event generation phase happen in serial, the last stages operate as a true pipeline in our current implementation. Events from the slicing stage are immediately sent to the vector clocking stage, which are immediately sent to the property checking stage. This differs from jPredictor were each stage is completely finished before the next begins and results in only reading each event from the logged trace once. This is a very important optimization, because it reduces the number of disk accesses proportionate to the number of pipelined stages by the length of the trace. Now we

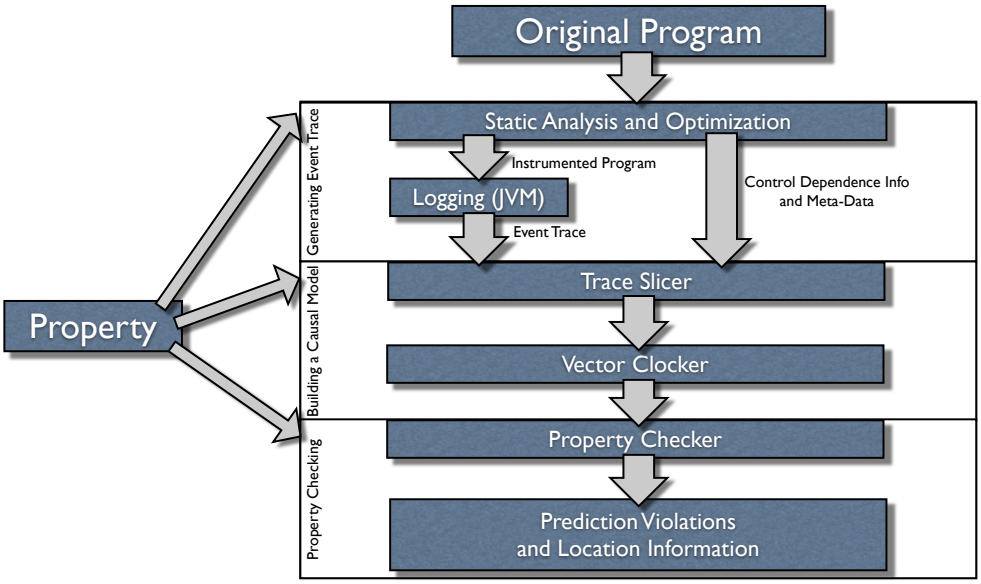



Figure 8.1: RV-Predict Pipeline

will give a brief overview of the stages involved.

The first stage in the pipeline, which is part of generating the event log, is to statically analyze the program to determine termination sensitive control dependence [36] and to apply instrumentation in the program. It is discussed fully in Section 8.3.1. The static analyzer is also where static optimizations to reduce the amount of logging at runtime occur. By reducing the amount of logging, it not only reduces the run time of the logged program, but also reduces the amount of time necessary for the rest of the pipeline due to far shorter traces. The property to be predicted is used to guide the instrumentation process because each property may need to instrument different events.

The second stage of the pipeline, which is also part of generating an event log, runs the instrumented program to collect a log. It is discussed in more detail in Section 8.3.2. A very important aspect of the logging phase is compression of the trace. Runs of an initial, compression-less, version of RV-Predict found that majority of time was spent writing the log to disk. By using fast compression, we drastically cut down on the amount of data written to disk. Another very important aspect is reverse logging because, as will be explained below, the next stage is trace slicing, which must operate in reverse program order. In jPredictor, logging occurred in program order, then the *entire* trace was read backward using random disk I/O. This is an incredible inefficiency in the system, that we hoped to reduce

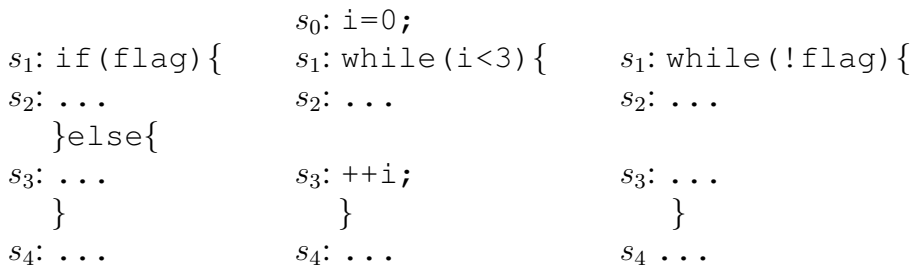



Figure 8.2: Control Scope Examples

using memory buffering. The only way that could be achieved was through either reversing the whole trace on disk, or performing reverse logging. Reverse logging is obviously preferable to reversing the whole trace, given that the length of a trace is only bounded by disk space.

The next stage is the trace slicer, part of building a causal model. As mentioned, the trace slicer operates in reverse, streaming through a log that was written out in reverse program order. The job of the trace slicer is to determine events of interest, as well as the events on which they depend. Section 8.4.1 explains this fully, but briefly, events of interest depend on the property in question. For race detection this involves every read and write of potentially shared variables. For generic properties, this means user defined events. For instance, for the SAFEENUM property first shown in Fig. 1.2 in Chapter 1, such an event would be the createE event. Slicing must operate in reverse because it cannot be know *a priori* which instructions events of interest are dependent on. In reverse it is straight-forward to tell which instructions contributed to a given event.

After trace slicing, is the vector clocking stage, also part of building a causal model, described fully in Section 8.4.2. The vector clocking stage is able to construct a causal model in order to find possible violations of properties that did not occur at runtime, but could occur in practice, given the dependences in the program. Traditional vector clock algorithms operate in a forward pass, but we developed a reverse algorithm so that there would be no need to reverse the trace after slicing. In the original jPredictor, the slices produced by the slicer had to be reversed for the vector clocking stage.

The last stage, the only one in the property checking phase, is dependent on the property to be checked, and is described in Section 8.5. It performs state exploration based on the causal model, while winnowing down the state space as much as possible.

## 8.3 Generating Event Logs

Generating event logs for RV-Predict is a two step process. In the first step various static analysis/optimizations are performed, and instrumentation is inserted into the program. In the second step, the program is run. The instrumentation in the program automatically generates event the event stream as the program runs.

The static analysis stage of RV-Predict is used to compute termination-sensitive control dependence [36], various meta data about the program (such as line numbers) used for error detection and reporting, instrumenting the program, and optimizing the code for logging purposes.

During the logging stage, the instrumented program is run on any Java virtual machine (JVM). The instrumentation automatically reverses and compresses the output trace as it is generated.

### 8.3.1 Static Analysis

All static analysis and program transformation is performed using the Soot java optimization framework [123]. Currently, the first step in static analysis is the generation of a *loop-peeled* program. Loop peeling is a process whereby one or more iterations of a loop are removed as straight line code. The purpose of this transformation is to limit the amount of logging within loops, that is we remove one (or more) iterations, and only instrument within the removed iterations. Rather than emit events for one million iterations of a given loop, for example, we may emit events for only the first five iterations. While this can cause us to miss potential violations, it vastly improves speed. For many types of properties, for example data races, the vast majority can be found in simply the first iteration of a loop. Currently, the loop peeler only removes the first iteration, but as future work we plan to make it fully tunable. The next section presents the loop peeling algorithm in full detail.

The next step of static analysis generates the termination-sensitive control dependence and some meta-data (in particular instruction line numbers). Terminal-sensitive control dependence finds what are known as *control scopes*. A statements $s_2$ is in the control scope of a statement $s_1$ if the execution of $s_2$ depends somehow on a choice at $s_1$. This becomes relevant during trace slicing, as we shall see, because if $s_2$ is in the control scope of $s_1$, any value that informs the decision at $s_1$ can introduce synchronization before $s_2$. For instance, if $s_1$ checks the value

of a flag variable set by another thread, $s_2$ cannot be executed unless that flag is set appropriately. This idea will be explained in more detail in Section 8.4.1. Fig. 8.2 shows three different examples of control scopes. In the first code snippet, $s_2$ and $s_3$ are in the control scope of $s_1$, but $s_4$ is not because $s_4$ executes regardless of the choice at $s_1$. In the second, $s_4$ is not in the control scope of $s_1$ because our termination analysis determines that the loop must always terminate. In the last example, however, $s_4$ *is* in the control scope of $s_1$ because the loop is not guaranteed to terminate. Its termination depends on the value of a variable that is presumably set in another thread.

The last step of static analysis is to add the actual instrumentation that will log events as the program runs. This is described in Section 8.3.2 because it is intimately related to the logging phase of RV-Predict.

### Loop Peeling

Fig. 8.3 shows a pseudocode description of the algorithm used for loop peeling. It omits several small details,[3] but the main gist of the algorithm is represented faithfully. Currently, it only peels one iteration, but we wish to make this a tunable feature as future work, perhaps even allowing random jumping between instrumented and uninstrumented iterations. Note that it is specific to the Java programming language, but could be applied to other languages with little modification; lines 1–2 of main simply iterate over each Method in the program. On line 5 we call the built-in loop finder in Soot [123] on a given Method, then on line 6 we loop through each loop found by the LoopFinder, calling the handleLoop function.

The handleLoop function is tasked with cloning the statements in a loop in order to provide the peeled iteration, patching references within the peeled loop, and cloning any Methods that are called within a loop body. Cloned Methods are marked so that no instrumentation will be added to them. This is a very important step because not only does having instrumentation inside of Methods called by uninstrumented code lose much of our benefit of loop peeling, but it can also result in traces that the trace slicer does not understand, e.g., the beginning of a Method that is never called. As is explained in the next section, we emit one event for a Method call, and then another event at the beginning of the called Method so that

[3] As two examples of details: we must ensure that we do not copy the back jumps of loops since we only wish to peel one iteration, not create two loops, and in order to properly clone methods, because of the inability to rename constructors in Soot, we add a parameter to all method clones that is typed with a class that we generate at instrumentation time.

```
Algorithm LoopPeeler(Program P)
function main(Program P)
1  for all Class c ∈ P do
2  ⋮ for all Method m ∈ c do
3  ⋮ ⋮ LoopFinder lf = loopFinderFor(m)
4  ⋮ ⋮ for all Loop l ∈ lf do
5  ⋮ ⋮ ⋮ handleLoop(l)
6  ⋮ ⋮ endfor
7  ⋮ endfor
8  endfor
function handleLoop(Loop l)
Locals : List⟨Statement⟩ loopClone,
         Map⟨Statement → Statement⟩ statementToClone
1  for all Statement s ∈ l do
2  ⋮ Statement clone = s.clone()
3  ⋮ loopClone.append(clone)
4  ⋮ statementToClone.put(s, clone)
5  ⋮ if s is a CallStatement
6  ⋮ ⋮ Method m = s.getCalledMethod()
7  ⋮ ⋮ Method clonedM = createUninstrumentableClone(m)
8  ⋮ ⋮ s.setCalledMethod(cloneM)
9  ⋮ endif
10 ⋮ s.addTag(NoInstrumentTag)
11 ⋮ patchReferences(loopClone, statementToClone)
12 ⋮ m.insertBefore(l.firstStatement(), loopClone)
13 endfor
function createUninstrumentableClone(Method m)
Local Method ret
1  ret.setParameterList(m.getParameterList().clone())
2  ret.addTag(NoInstrumentTag)
3  for all Statement s ∈ m do
4  ⋮ Statementclone = s.clone()
5  ⋮ if s is a CallStatement
6  ⋮ ⋮ Method cloneM = createUninstrumentableClone(m)
7  ⋮ ⋮ clone.setCalledMethod(cloneM)
8  ⋮ endif
9  ⋮ ret.insertAtEnd(clone)
10 endfor
11 return ret
function patchReferences(statements, statementToClone)
1  for all Statement s ∈ statements do
2  ⋮ if sis a BranchStatement
3  ⋮ ⋮ s.setTarget(statementToClone.get(s.getTarget()))
4  ⋮ endif
5  endfor
```

Figure 8.3: Loop Peeling algorithm

the trace slicer may patch the value used within the Method to the argument passed to the Method, if that value were a parameter. This is necessary to achieve the correct dependence information; without it, additional false positives may result because the dependence chains would not be fully realized (see Section 8.4.1). As we can see on line 1 of handleLoop, the Method loops over each Statement in the loop. On line 2 we create a clone of each Statement, we do this because we wish to alter the original statements distinctly from the clones. On line 3, we add each cloned Statement to a list of Statements that will be inserted before the loop on line 12, providing our peeled iteration. On line 4, we add a mapping from each Statement $s$ to its clone, this allows us to correctly patch jumps in the peeled loop because jumps are targeted to specific Statements, which we can see in the call to patchReferences on line 11. Lines 5-9 handle the case where $s$ is a call to a Method. If $s$ is a call to a Method, we need to clone that Method, as mentioned, so that the clone can be uninstrumented. Because it is the clone that will be uninstrumented, we call the cloned Method from the *original* Statement $s$, rather than the Statement *clone* (line 9). On line 10 we add a NoInstrumentTag to $s$, so that the instrumentor will not add instrumentation for that Statement, in the case that it normally would (see Section 8.3.2).

The createUninstrumentableClone function simply creates a new Method, *ret*, iterates over all the Statements in $m$ (line 3), and adds clones of those Statementss to *ret* (line 4). It must, however, clone the parameter list from $m$ as can be seen on line 1, add the NoInstrumentTag to the Method as seen on line 2, and recursively clone any Method called by the Method to be cloned, as can be seen on lines 5–6. Adding the NoInstrumentTag to the Method rather than the Statements it contains makes the instrumentation phase faster, as it can simply skip a whole Method, if it is tagged.

The last function, patchReferences, loops over every Statement in the list of Statements it is passed, and replaces the target Statement of any BranchStatement with its clone, which is specified in the *statementToClone* map that is passed as an argument. The patching functionality must happen as a separate pass, as shown, so that all Statements in the loop have clones.

### 8.3.2 Logging

Here we present some of the engineering choices used for the logging code, as well as the different events that are logged. For compression we use a Java imple-

mentation of Ziv-Lempel (LZ) encoding [129], without the subsequent Huffman encoding [70] used by the Deflate algorithm (zlib). Our original attempt used the Deflate algorithm [43], but we found the faster compression time of LZ resulted in an overall faster logging experience. While some compression is necessary to keep RV-Predict from being completely I/O bound, the extra compression of Deflate did not justify the increased compression/decompression times, especially in cases where solid state drives are used instead of rotational hard drives.

All events are serialized using Google's protocol buffers [103]. Protocol Buffers are a way of encoding structured data in an efficient yet extensible format, and Google uses protocol buffers for almost all of its internal RPC protocols and file formats. It is much more efficient and easier to use than alternatives such as XML or Java's serialization.

In order to reverse the traces as logging occurs, we keep a buffer of events in reverse. The entire buffer is serialized at once into separate files. Currently, the buffer is 100,000 events, which we have determined produces good run times. Many more events results in high memory pressure and slower logging, many fewer results in larger compression overhead and less compression of the trace. When the slicing stage reads in the trace, it reads the files in reverse order. For future work we intend to move the separate traces files into entries in the database that we use to back the working sets for the slicer, vector clock, and property checker.

While logging for race detection, we also run the Racer [25] algorithm for data race detection to locate potential data races. We then use these potential data races to determine which variables we should perform data race detection. Racer produces numerous false positives; the RV-Predict race detection algorithm narrows these down to the actual occurring races.

Next, we list and explain the different events that are logged for RV-Predict:

- Lock and Unlock: These vents correspond to the beginning and end of synchronized blocks in Java. In Soot, these are represented as the entermonitor and exitmonitor instructions when the occur in method bodies. We also emit lock and unlock events at the beginning and end of synchronized methods. These are necessary only for race detection, where they are used for computing lock sets (see Section 8.5.1).

- Branch: We must log branches and jumps to correctly compute dependences induced by termination-sensitive control dependence. If a statement of interest is control dependent on a branch $b$, then it is dependent on all definitions

that contribute to the boolean value that determines the direction of $b$.

- CallStatement: We must log the calls of methods so that we can correctly relate the parameter values within a method with the argument passed to the method, as mentioned in Section 8.3.1.

- Constructor: This corresponds to the end of constructors. We must log constructors separately because the end of a constructor is a synchronization point in Java. There can be no intereleavings such that a modification of an object $o$ by a normal method occurs before the end of the constructor that constructed $o$.[4] For instance, there can be no race between a modification in a normal method and a modification within $o$'s constructor.

- Method: This corresponds to the begging of a method call, in the callee context. This allows for patching the parameters with the arguments, as mentioned in Section 8.3.1.

- Accesses: We must log reads and writes to access so that we can properly link values through data dependence. This is one of the most expensive parts of logging, and is one of the biggest reasons that the loop peeling optimization works so well.

- ImpureCallStatement: We log impure calls separately from normal calls. Impure calls are calls to external methods that we do not know, *a priori*, to not modify their arguments. For impure methods we assume that modifications to all arguments occur.

- Wait, Notify, Start, Join, etc: Java has a number of defined synchronization constructs such as wait, notify, start, and join. Each is given its own event because they must be handled separately.

- MOP Events: We have a special logging class for MOP events that properly keeps track of the parameters to the events, so that they may be properly serialized for the JavaMOP monitoring used when predicting generic properties.

- Finalizer: This corresponds to the beginning of finalizers. We must log finalizers separately, because Java guarantees synchronization with finalizers. There can be no interleavings such that modifications to an object $o$ occur

[4] Unless the object is leaked to another thread by the constructor.

outside of a finalizer happen after initiation of the finalizer. For instance, there can be no race between a modification in a normal method and a modification within $o$'s finalizer.

## 8.4 Building a Causal Model

Here we describe our technique for extracting from an execution trace of a multi-threaded system the sliced causality relation corresponding to some property of interest $\varphi$. Our technique is *offline*, in the sense that it takes as input an already generated execution trace (see Fig. 8.1); that is because causal slicing must traverse the trace backwards. Our technique consists of two steps: (1) all the irrelevant events (those which are neither property events nor events on which property events are dependent) are removed from the original trace, obtaining the *($\varphi$)-sliced trace*; and (2) a *vector clock (VC)* based algorithm is applied on the sliced trace to capture the sliced causality partial order.

### 8.4.1 Extracting Slices

Our goal here is to take a trace $\xi$ and a property $\varphi$, and to generate a trace $\xi_\varphi$ obtained from $\xi$ filtering out all its events which are irrelevant for $\varphi$. When slicing the execution trace, one must nevertheless keep all the property events. When slicing for race detection, we create a separate slice for each shared variable that is determined by the Racer algorithm to have a potential race, that is the property events for each slice are only the reads and writes to a specific variable (see Section 8.5.1). For generic properties we take the user defined events as the property events. While it would be possible to create separate slices for each parameter instance, we prefer to create one slice with all the property events, and to allow the monitor produced by JavaMOP to do its own parametric slicing (see Section 8.5.2). While one could conceivably predict multiple properties at once, and create a separate slice for each property, we currently only allow predicting one property at a time, we will add that ability as future work, however.

Moreover, one must also keep any event $e$ with $e\ (\sqsubset_{ctrl} \cup \sqsubset_{data})^+\ e'$ for some property event $e'$. This can be easily achieved by traversing the original trace backwards, starting with $\xi_\varphi$ empty and accumulating in $\xi_\varphi$ events that either are property events or have events depending on them already in $\xi_\varphi$. One important aspect, as mentioned earlier, is patching values that are passed as arguments to

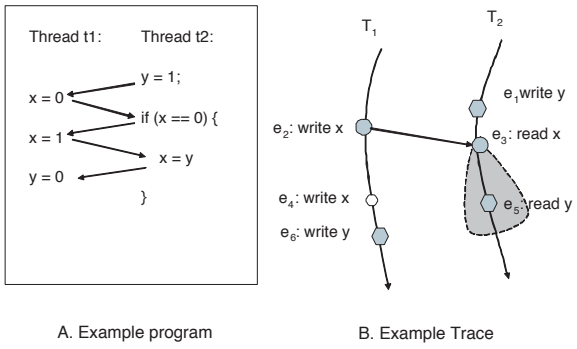



Figure 8.4: Example for relevance dependence

methods. If a property event is dependent on a value that is a parameter to a method, we must patch that value to the passed value in the calling context. For instance, if a property event depends on the value in the variable $x$, which was an argument to the current method, and in the calling context $y$ was passed into the argument position for $x$, we must add $y$ to the tracked values as slicing continues in the calling context. This process must be repeated transitively.

If one were to attempt slicing on a forward trace, one must assume that *all* events $e \ (\sqsubset_{ctrl} \cup \sqsubset_{data})^+ \ e'$, only determining which ones may be removed from the set of considered events at the end of the trace. This would be far too expensive, both in terms of time and space. In fact, it would require so much space (on the order of the length of the trace), that the sets of event would need by kept in disk storage. One can employ any off-the-shelf analysis tool for data- and control-dependence; e.g., RV-Predict uses termination-sensitive control dependence [36], as described in Section 8.3.1. We do not present the slicing algorithm in full, as we do with loop peeling and the property checking algorithm, because it is too complex to fit nicely within the chapter, and because program slicing is a well known topic. While our slicing is different than static program slicing such as in [67], we feel it is sufficiently close to allow for an informal description.

To understand the process (intuitively), consider the example in Fig. 8.4, threads $T_1$ and $T_2$ are executed as shown by the solid arrows (A), yielding the event sequence "$e_1$, $e_2$, $e_3$, $e_4$, $e_5$, $e_6$" (B). Suppose the property to check refers only to $y$; the property events are then $e_1$, $e_5$, and $e_6$. Events $e_2$ and $e_3$ are immediately marked as relevant, since $e_2 \sqsubset_{data} e_3 \sqsubset_{ctrl} e_5$. If only closure under control- and data-dependence were used to compute the relevant events, then $e_4$ would appear to be irrelevant, so one may conclude that "$e_2$, $e_6$, $e_1$, $e_3$, $e_5$" is a sound permutation;

there is, obviously, no execution that can produce that trace, so one reported a false alarm if that trace violated the original property on $y$. Consequently, $e_4$ is also a relevant event and $e_3 \sqsubset_{rlvn} e_4$.

Unfortunately, one backwards traversal of the trace does not suffice to correctly calculate all the relevant events. Reconsider Fig. 8.4. When the backward traversal first reaches $e_4$, it is unclear whether $e_4$ is relevant or not, because we have not seen $e_3$ and $e_2$ yet. Thus a second scan of the trace is needed to include $e_4$. Once $e_4$ is included in $\xi_\varphi$, it may induce other relevance dependencies, requiring more traversals of the trace to include them. This process would cease only when no new relevant events are detected and thus resulting sliced trace stabilizes. If one misses relevant events like $e_4$ then one may "slice the trace too much" and, consequently, one may produce false alarms. Because at each trace traversal some event is added to $\xi_\varphi$, the worse-case complexity of the sound trace slicing procedure is square in the number of events. Since execution traces can be huge, on the order of billions of events, any trace slicing algorithms that is worse than linear may easily become prohibitive. For that reason, RV-Predict traverses the trace only once during slicing, thus achieving an approximation of the complete slice that can, in theory, lead to false alarms. However, experiments in [37] show that this approximation is actually very precise in practice, finding no false alarms, and we have not found any false alarms in any of our experiments since.

### 8.4.2 Vector Clocking

Vector clocks [84] are routinely used to capture causal partial orders in distributed and concurrent systems. A *VC*-based algorithm was presented in [112] to encode a conventional multithreaded-system "happen-before" causal partial order on the unsliced trace. We next adapt that algorithm to work on our sliced trace and thus to capture the sliced causality. Recall that a vector clock (*VC*) is a function from threads to integers, $VC : T \to Int$. We say that $VC \geq VC'$ iff $\forall t \in T, VC(t) \geq VC'(t)$. Traditional vector clocking algorithms operate on forward traces, and concern themselves with the $\max$ of multiple *VC*. Because our vector clocking stage operates in reverse, we are concerned instead with the $\min$. The $\min$ function on *VCs* is defined as: $\min(VC_1, ..., VC_n)(t) = \min(VC_1(t), ..., VC_n(t))$ (similar to how $\max$ is defined in [112]).

Before we explain our *VC* algorithm, let us introduce our event and trace notation. An *event* is a mapping of *attributes* into corresponding *values*. One event

can be, e.g., $e_1 : (\textit{counter} = 8, \textit{thread} = t_1,\ \textit{stmt} = L_{11},\ \textit{type} = \textit{write},\ \textit{target} = a,\ \textit{state} = 1)$, which is a write on location $a$ with value 1, produced at statement $L_{11}$ by thread $t_1$. One can include more information into an event by adding new attribute-value pairs. We use $\textit{key}(e)$ to refer to the value of attribute *key* of event $e$. To distinguish different occurrences of events with the same attribute values, we add a designated attribute to every event, *counter*, collecting the number of previous events with the same attribute-value pairs (other than the *counter*). The trace for the vector clocking step is the $\varphi$-sliced trace $\xi_\varphi$ obtained in Section 8.4.1. Recall that the trace arrives one at a time, directly from the slicing stage, rather than read from disk during the slicing stage, as in jPredictor.

Intuitively, vector clocks are used to track and transmit the causal partial ordering information in a concurrent computation, and are typically associated with elements participating in such computations, such as threads, processes, shared variables, messages, signals, etc. If *VC* and *VC*′ are vector clocks such that $\textit{VC}(t) \geq \textit{VC'}(t)$ for some thread $t$, then we can say that *VC'* has newer information about $t$ than *VC*. In our *VC* technique, every thread $t$ keeps a vector clock, $\textit{VC}_t$, maintaining information about all the threads obtained both locally and from thread communications (reads/writes of shared variables). Every shared variable is associated with two vector clocks, one for writes ($\textit{VC}_x^w$) used to enforce the order among writes of $x$, and one for all accesses ($\textit{VC}_x^a$) used to accumulate information about all accesses of $x$. They are then used together to keep the order between writes and reads of $x$, iff $x$ is significant to the property in question. Every property event $e$ found in the analysis is associated a *VC* attribute, which represents the computed causal partial order. We next show how to update these VCs when an event $e$ is encountered during the analysis, the third case can overlap the first two; if so, the third case will be handled first. For the sake of simplicity, we do not include rules for wait, notify, and constructors/finalizers here,[5] but this should give a firm grasp of the algorithm:

1. $\textit{type}(e) = \textit{write}$, $\textit{target}(e) = x$, $\textit{thread}(e) = t$ (the variable $x$ is written in thread $t$) and $x$ is a shared variable. In this case, the write vector clock $\textit{VC}_x^w$ is updated to reflect the newly obtained information; since a write is also an access, the access *VC* of $x$ is also updated; we also want to capture that $t$ committed a causally irreversible action, by updating its *VC* as well: $\textit{VC}_t(t) \leftarrow \textit{VC}_t(t) - 1$, $\textit{VC}_t \leftarrow \textit{VC}_x^a \leftarrow \textit{VC}_x^w \leftarrow \min(\textit{VC}_x^a, \textit{VC}_t)$.

[5]These constructs require separate, special sets of vector clocks.

2. *type*$(e)$ = *read*, *target*$(e) = x$, $thread(e) = t$ (the variable $x$ is read in $t$), and $x$ is a shared variable. Then the thread updates its information with the write information of $x$ (we do not want to causally order reads of shared variables!), and $x$ updates its access information with that of the thread: $VC_t(t) \leftarrow VC_t(t) - 1$. $VC_t \leftarrow \min(VC_x^w, VC_t)$, $VC_x^a \leftarrow \min(VC_a^x, VC_t)$.

3. $e$ is a property event and $thread(e) = t$. Then $VC_t(t)$ is decreased to capture the intra-thread total ordering: $VC_t(t) \leftarrow VC_t(t) - 1$.

4. *type*$(e)$ = *start*, *target*$(e) = t_1$, *thread*$(e) = t_2$. Here $t_1$ is the thread started by the *start* event, while $t_2$ is the thread in which start is called. Calling start implies that any event in the new thread must occur after the call to start, thus we have $VC(t_2)_{t_2} \leftarrow VC(t_2)_{t_2} - 1$, $VC_{t_1} \leftarrow \min(VC_{t_1}, VC_{t_2})$ (keep in mind that start will actually be the last event for $t_1$, since traces are reverse program order).

5. *type*$(e)$ = *join*, *target*$(e) = t_1$, *thread*$(e) = t_2$. Here $t_1$ is the thread started by the *start* event, while $t_2$ is the thread in which start is called. Calling join implies that any event in the calling thread, $t_2$ must occur after events in the joined thread, $t_1$, thus we have $VC(t_2)_{t_2} \leftarrow VC(t_2)_{t_2} - 1$, $VC_{t_1} \leftarrow VC_{t_2}$ (keep in mind that join will actually be the first event for $t_1$, since traces are reverse program order).

## 8.5 Property Checking

Currently, as mentioned, there are two separate prediction algorithms in RV-Predict. We have a generic algorithm for predicting properties specified using JavaMOP, and we have a specialized algorithm that is more efficient for data races. We will start with the algorithm for data races because it is simpler, and thus easier to understand, while the generic algorithm is a generalization of that algorithm. Understanding the data race detection algorithm eases the understanding of the generic prediction algorithm. An important point with both algorithms is that the deal with *only* property events. For race detection this means all reads/writes to the shared variable of interest, for generic properties this means the user defined JavaMOP events. Because of this, even though the algorithm for generic prediction is more complicated, it generally takes less time, as the traces are far shorter. The

```
Algorithm DataRaceDetection(Trace τ, List⟨ThreadIterator⟩ threads)
Global : Set⟨RaceSearchState⟩ RST
Initialization : ∀ t1, t2 ∈ threads s.t. t1 ≠ t2
                    RST ← RST ∪ {createRaceSearchState(t1, t2)}
function main
1  for RaceSearchState rst ∈ RST do
2  ⋮ if rst.t1.VC > rst.t2.VC
3  ⋮ ⋮ RST.add(createRaceSearchState(decrement(rst.t1), rst.t2))
4  ⋮ elseif rst.t1.VC < rst.t2.VC
5  ⋮ ⋮ RST.add(createRaceSearchState(rst.t1, decrement(rst.t2)))
6  ⋮ else
7  ⋮ ⋮ if rst.t1.event is a WriteEvent || rst.t2.event is a WriteEvent
8  ⋮ ⋮ ⋮ if rst.t1.lockSet ≠ rst.t2.lockSet
9  ⋮ ⋮ ⋮ ⋮ reportRace()
10 ⋮ ⋮ ⋮ endif
11 ⋮ ⋮ endif
12 ⋮ ⋮ RST.add(createRaceSearchState(rst.t1, decrement(rst.t2)))
13 ⋮ ⋮ RST.add(createRaceSearchState(decrement(rst.t1), rst.t2))
15 ⋮ endif
16 endfor
```

Figure 8.5: Data Race Detection algorithm

other events added due to control/data dependence were *only* for the purposes of assigning vector clocks to events in the vector clocking stage.

### 8.5.1 Data Race Detection

The basic idea of race detection is simple: check for accesses to the same variable with incomparable *VCs*. However, it is easy to note that this has quadratic worst case complexity, because each access must be compared against every other access. Clearly, when billions of accesses may occur in a trace, this is unacceptable. Not only would this be unbearable slow, but it would be impossible to even fit the accesses in memory to perform the comparisons.

To alleviate this, as well as to make it more easy to deal with streaming to and from the disk when memory is overfull, we use the idea of a window of comparisons, ignoring pairs of events that trivially cannot have incomparable vector clocks. If at some point we note the second access, $a_2^{T_1}$ in thread $T_1$ must occur after the fifth access, $a_5^{T_2}$, in thread $T_2$ we know that we do not need to check the $a_2^{T_1}$ against any further accesses in thread $T_2$ because all accesses in a given thread must be totally ordered (and the traces are backwards).

To implement this we use a *Set*.[6] of RaceSearchStates Fig. 8.5 shows a simplified pseudocode description for our algorithm, which is inspired by the idea of continuations in functional programming. Each search state abstracts the notion of checking accesses in two threads. Each RaceSearchState keeps an iterator to the list of accesses representing one of its two given threads (ThreadIterator); by this stage of the RV-Predict pipeline the only remaining events are accesses, as the vector clocking stage filters out non-property events as it adds vector clocks. The algorithm begins by keeping search states for each pair of threads passed to the algorithm in the set *RST*, as can be seen in the initialization line. Each state is advanced by considering the accesses pointed to by each of its iterators. If the vector clocks of the two accesses in question are ordered, only one of the iterators is advanced, for example, if the access in thread $t$ of the search state must take place before the access in thread $t'$, the iterator pointing to the access from thread $t'$ is decremented (keep in mind the traces are in reverse program order), as can be seen on lines 2–6. On the other hand, if the iterators are incomparable, two new search states are added to the set. One state where one iterator is decremented, and one where the other iterator is decremented, as can be seen on lines 11–13.

### 8.5.2 Generic Property Checking

As mentioned the race detection algorithm can be extrapolated to generic property prediction. There are a few caveats: the iterators of the search states point to streams of monitoring events like those described in Section 8.3.2 rather than accesses to shared variables, and each search state keeps an iterator to every thread in the program in a list. Each search state, additionally, keeps a reference to a monitor provided by JavaMOP. The provided monitors cannot provide the "good-prefix" matching allowed by JavaMOP because our algorithm works in reverse. We instead only report violations based on the last state of the monitor when the end (beginning) of the trace is reached. For all logics but SRS, JavaMOP is able to easily create a monitor for prediction by reversing the finite state machine or context-free grammar, similar to how we compute coenable sets using enable set algorithms (see Chapter 3). For future work, we intend to add a forward prediction pass that will actually reverse the trace, when good prefix matching is required. For SRS, the user must provide an SRS that will work for backward traces.

Fig. 8.6 shows pseudocode for the generic property prediction algorithm. It is

[6]We must use a set to avoid duplicate search states, or the algorithm can quickly explode.

```
Algorithm GenericPropertyPrediction(Trace τ, List⟨ThreadIterator⟩ threads)
Global : Set⟨SearchState⟩ ST
Initialization : ST ← {createSearchState(sort(threads), initialMonitor)}
function main
Local : int i
1  for SearchState st ∈ ST do
2  ⋮ if st.threadIterators.get(0).VC > st.threadIterators.get(1).VC
3  ⋮ ⋮ List⟨ThreadIterator⟩ nextThreads = st.threadIterators.clone()
4  ⋮ ⋮ Monitor nextMonitor = st.monitor.clone()
5  ⋮ ⋮ nextMonitor.process(st.threadIterators.get(0).event)
6  ⋮ ⋮ ST.add(createSearchState(bubbleRight(nextThreads, 0), nextMonitor))
7  ⋮ else
8  ⋮ ⋮ for i = 2 to threads.size() − 1 do
9  ⋮ ⋮ ⋮ if st.threadIterators.get(0).VC > st.threadIterators.get(i).VC
10 ⋮ ⋮ ⋮ ⋮ break
11 ⋮ ⋮ ⋮ endif
12 ⋮ ⋮ endfor
13 ⋮ ⋮ createPermutedStates(st, i)
14 ⋮ endif
15 endfor
function createPermutedStates(SearchState st, int i)
Local : int j
1  for j = 0 to i − 1 do
2  ⋮ List⟨ThreadIterator⟩ nextThreads = st.threadIterators.clone()
3  ⋮ Monitor nextMonitor = st.monitor.clone()
4  ⋮ nextMonitor.process(st.threadIterators.get(j).event)
5  ⋮ ST.add(createSearchState(bubbleRight(nextThreads, j), nextMonitor))
6  endfor
function bubbleRight(List⟨ThreadIterator⟩ threads, int i)
Locals : int j, ThreadIterator it
1  it = threads.remove(i)
2  decrement(it)
3  for  j = i + 1 to threads.size() − 1 do
4  ⋮ if it.VC ≮ threads.get(j).VC
5  ⋮ ⋮ break
6  ⋮ endif
7  endfor
8  threads.insertBefore(j, it)
9  return threads
```

Figure 8.6: Generic Property Checking algorithm

important to maintain the ordering, monotonically decreasing by vector clock, in the thread iterator list. By keeping the threads sorted by vector clock, we drastically reduce the number of necessary comparisons. Due to the need to keep the thread iterator list sorted, it is implemented as a linked list. We will use the terminology right to mean further in the list, and thus incomparable or lesser *VCs* than the current position in the list.

We initialize the algorithm with one SearchState in the set *ST*. The createSearchState function simply creates a new SearchState with the passed list of ThreadIterators and Monitor. In the initialization we begin with the initialMonitor.

The main function is the driver of the algorithm, it loops over all the SearchStates in *ST* on line 1. For each SearchState it compares the first two elements of the list of ThreadIterators. Because the list is soft monotonically decreasing, there are only two possibilities: the first *VC* in the list will be greater than the second, or it will be incomparable. The short case, from lines 2–6, the second is less than the first. In that case, we send the event pointed to by *st.threadIterators*.get(0) to a clone of the Monitor in *st*, that will be put into the new SearchState that we are generating, on line 6. We also call the bubbleRight function with a clone of the ThreadIterators of *st*, which we pass to the creation of the next SearchState. The bubbleRight function, which will be explained in more detail below, is responsible for decrementing the ThreadIterator pointed to by the passed integer, and pushing it to the right in the list as long as its *VC* is less than the *VC* of the next ThreadIterator to the right, which may occur due to the decrementing. This is a valid operation: we may decrement the ThreadIterator at the current position without violating the causal model. The bubbleRight function also ensures that we maintain our sorting.

The second case, lines 7–14, is complicated enough to require its own function for ease of understanding: createPermutedStates. This case occurs when the *VC* of *st.threadIterators*.get(0) is incomparable to that of *st.threadIterators*.get(1). Because incomparability is transitive with monotonically decreasing sorting, it is possible that several of the next right ThreadIterator's will he incomparable. The loop on lines 8–12 finds the range of incomparable *VCs*.

The createPermutedStates function, creates SearchStates for an entire range of ThreadIterators with incomparable *VCs* by generating one Search state where each of the ThreadIterators is decremented. This represents the idea that any of these threads could be chosen to go next when predicting thread interleavings because they are all incomparable. As with the greater than case, we must allow the bubbleRight function to handle the decrementing and inserting, in case decre-

menting the given ThreadIterator results in a ThreadIterator that is less than its neighbor to the right.

Finally, as mentioned, the bubbleRight function decrements the ThreadIterator pointed to by the passed integer $i$, and pushes it to the right in the passed list of ThreadIterators. On line 1, it first removes the ThreadIterator, *it*, which $i$ points to, from the list, so that we do not have two copies in the list. One line 2, it decrements the *it*. The loop from lines 3 to 7 finds the first spot where it is safe to insert *it* without upsetting the sort. On line 8, *it* is inserted in the proper place found by the loop, and on line 9 the list of ThreadIterators is returned.

This algorithm is exponential, but it minimizes repeated work due to the sorting and set collapsing. Unfortunately, any algorithm to search the state space will be exponential, but on the positive: the trace slices generated for generic properties tend to be short, particularly with the loop peeling optimization.

## 8.6 Experiments

Fig. 8.7 summarizes the differences in real time and disk usage between the original jPredictor system first presented in [37] and RV-Predict for prediction stages only[7] for race prediction as measured on a system with two quad core Xeon E5430 processors running at 2.66GHz and 16 GB of 667 MHz DDR2 memory running Redhat Linux. On very small examples jPredictor occasionally outperforms RV-Predict, but on anything substantial RV-Predict is a vast improvement. Account, elevator, and tsp are actual programs used to benchmark parallel systems. Huge, medium, small, and the mixed locks examples are microbenchmarks that we designed to test particularly difficult aspects of race detection, such as millions of accesses to the same shared variable in huge.

While the results above are compelling, showing that RV-Predict is a vast improvement over jPredictor, we wish to show that RV-Predict is usable on larger programs. We also wish to know exactly which parts of RV-Predict are currently responsible for the most overhead.

For our experiments on the different causes of overhead we we used an Alienware m14x laptop with an Intel® Core™ i7-2670QM with a peak frequency of 2985.87 MHz. It has 8GB of 1600 MHz DDR3 ram, and a Samsung® 830 Series 256GB SSD. Unfortunately, while the 830 series is a SATA-III drive, the

[7] These numbers are from an earlier version of RV-Predict that did not have loop peeling.

| | | jPredictor | | RV-Predict | |
|---|---|---|---|---|---|
| Name | Input | Real Time | Disk Usage | Real Time | Disk Usage |
| account | - | 0:02.07 | 236K | 0:04.31 | 360K |
| elevator | - | 5:55.29 | 63M | 1:20.31 | 864K |
| tsp | map4 2 | 5:30.87 | 16M | 1:33.44 | 744K |
| tsp | map5 2 | 10:10.19 | 17M | 2:20.95 | 868K |
| tsp | map10 2 | 8:25:04.00 | 442M | 29:27.13 | 2.8M |
| huge | - | crash | crash | 0:42.22 | 13M |
| medium | - | crash | crash | 0:06.12 | 840K |
| small | - | crash | crash | 0:05.99 | 292K |
| mixedlockshuge | - | 8:13:40.00 | 250M | 0:13.95 | 2.9M |
| mixedlocksbig | - | 5:44.89 | 25M | 0:07.03 | 496K |
| mixedlocksmedium | - | 0:08.92 | 2.7M | 0:07.25 | 308K |
| mixedlockssmall | - | 0:05.46 | 1.5M | 0:05.67 | 296K |

Figure 8.7: jPredictor Vs. RV-Predict

| Benchmark | Total Time | Soot (SA) | Static Analysis | Program Run | Soot (Prediction) | Prediction |
|---|---|---|---|---|---|---|
| account×5 | 24.68 | 2.354 | 10.779 | 0.41 | 9.863 | 1.274 |
| simple | 22.972 | 2.343 | 10.193 | 0.29 | 9.591 | 0.554 |
| elevator 1 | 65.906 | 2.298 | 11.416 | 16.13 | 9.651 | 26.411 |
| 2 | 59.18 | 2.345 | 11.652 | 15.167 | 9.574 | 20.442 |
| 3 | 57.503 | 2.298 | 11.265 | 13.235 | 9.344 | 21.361 |
| 4 | 59.534 | 2.367 | 11.534 | 14.2 | 9.415 | 22.018 |
| 5 | 55.749 | 2.287 | 11.347 | 14.137 | 9.682 | 18.296 |

Figure 8.8: RV-Predict Stage Statistics (time in seconds)

Alienware BIOS limits the serial-ATA bus to SATA-II speed for stability reasons with some laptops.

Fig. 8.8 shows results for the different stages of the pipeline on three benchmarks: the elevator benchmark from our previous test, a version of the account benchmark from the previous test that uses 5× more accounts, as well as a benchmark we call simple, which creates two threads that access a shared memory location, causing a data race. The numbers for account and simple are stable, so we show only one run. For elevator we show the results from five trials because it takes random amounts of time to run due to using random scheduling of elevator threads.[8] The stages Soot (SA) and Soot (Prediction) are the times it takes to load Soot and the class *scene* for static analysis and prediction, respectively. The

[8]The elevator schedule is so random, in fact, that it will deadlock occasionally.

scene is the term Soot uses to refer to all the classes loaded into Soot. For our experiments this includes the programs themselves, as well as some of the libraries used. One point of interest is that the amount of time to load Soot tends to be consistent throughout the experiments, as is the time for static analysis. While elevator is quite a bit larger than account or simple, the change in static analysis time is rather low, denoting that quite a bit of the time is constant overhead. For future work we will test other Java compiler frameworks to see if we can improve upon static analysis and compiler framework load time. Also interesting is that, while prediction time is definitely proportional to the run time of the instrumented program (Program Run column), it is not always completely consistent. For instance, trail 3 of elevator has a longer prediction time than trial 2, which had a shorter program run. Part of this can attributed to random noise, but there is also a good chance that the shorter run actually induced a larger causal model space.

For larger programs we focused on three of the benchmarks in JavaGrande, and only looked at the time for the Prediction stage. For JGFLUFactorization-BenchSizeA with 4 threads, which does concurrent LU Factorization of matrices, four of the six race candidates take less than 10 seconds, while the other two, lufact.TournamentBarrier.isDone and lufact.TournamentBarrier.maxBusyIter take 1286 and 3434 seconds, respectively. The entire prediction phase takes 4783 seconds; that is one hour and 19 minutes. This points to an inefficiency in the prediction phase that we need to amend, but it may simply be impossible due to the exponential nature of causal model exploration. We intend to improve performance for particularly large causal models as future work. What is especially interesting is that lufact.TournamentBarrier.maxBusyIter takes 3434 seconds with a trace only 2.1 million events long. However, for JGFSeriesBenchSizeA with 4 threads we manage to predict for each variable in 400 seconds despite a logged trace length of *100 million* events. Overall, JGFSeriesBenchSizeA took 1846 seconds for prediction. This tells us that the length of a trace is not a good predictor for the complexity of the causal model. Our last benchmark, JGFSORBenchSizeA, took only 117 seconds to predict despite traces of 5 million events.

## 8.7 Chapter Related Work

There are several other approaches aimed at detecting potential concurrency errors by examining particular execution traces. Some approaches attempt to verify

general purpose properties [110, 112], including temporal ones, and are inspired from debugging distributed systems based on Lamport's *happens-before* causality [84]. Other approaches work with particular properties, such as data-races and/or atomicity. [108] introduced the first lock-set based algorithm to detect data-races dynamically, followed by many variants aiming at improving its accuracy. For example, an ownership model was used in [124] to achieve a more precise race detection at the object level. [97] combines lock-sets with happen-before. [52] provides a race detector that is both efficient and precise by switching between what they call *epochs* and vector clocks as necessary. However, their technique works only for race detection, not generic properties. We intend to look into modifications of their technique for generic property detection, as the performance gains would greatly improve the performance of RV-Predict. Numerous other race-detection techniques have been introduced, but we feel these are only tangentially related to this work.

Previous efforts tend to focus on either soundness or coverage: those based on happens-before try to be sound, but have limited coverage over interleavings, thus missing errors; lock-set based approaches have better coverage but suffer from false alarms. Our technique aims to improve coverage without giving up soundness or genericity of properties. The only previous system to have the same features is jPredictor [34, 37], on which RV-Predict is based. As mentioned, jPredictor does not work on real sized programs due to the very limited working sets it can handle, additionally, it offers no algorithm for generic property detection.

## 8.8 Chapter Conclusion

This chapter presents the RV-Predict predictive analysis system. It is able to *predict* general purpose properties that may arise in thread interleavings that do not occur at runtime, based simply on one execution of a program by building a *causal model* and exploring the interleavings induced by that model. RV-Predict is based on the ideas of jPredictor, but jPredictor simply did not work for realistic programs. Additionally, jPredictor had only algorithms for race detection and atomicity violations, while RV-Predict has an algorithm for generic property prediction that avoids repeating work. There is a large amount of future work the make RV-Predict into a reliable and efficient system. A suitable replacement for vector clocks, where possible, such as in [52] is a primary goal. We also intend to reduce

the amount of instrumentation necessary to generate the causal model. There is also a performance degradation for particularly large causal models that we need to address. Overall, however, RV-Predict demonstrates the efficacy of Runtime Verification in a predictive domain.

# Chapter 9

# Conclusion

With an ever increasing emphasis on software in our daily lives, improving the reliability of software becomes an ever more important issue. Runtime Verification is a quickly growing technique for providing many of the guarantees of formal verification, which helps improve software reliability. However, unlike formal verification, Runtime Verification achieves its goals in a manner that is scalable. The purpose of the work in this thesis is to give rise to *efficient*, *expressive*, and *effective* runtime verification. Every item in this thesis hopes to improve on or more of these facets. All work in this thesis, save for RV-Predict, is implemented in the Monitoring Oriented Programming (MOP) framework.

Chapter 2 introduces the MOP framework, discussing the relationships between the two current instantiates of MOP, JavaMOP and BusMOP. It goes to show that one general Runtime Verification framework is efficacious across two separate domains: Java programs and hardware hardware/software systems.

Chapter 3 gives a more in depth exposition on JavaMOP in particular. It showcases several techniques that are able to vastly improve the efficiency of parametric monitoring. It also introduces suffix monitoring and various parameter binding modes, that increase the expressivity of monitoring.

Chapter 4 goes into detail on BusMOP and the two differing environments in which it has been applied: PCI Bus traffic monitoring, and System on a Chip (SoC) design. The work on BusMOP shows that Runtime Verification is effective for hardware and hardware/software systems. The two different contexts in which it was applied lend further validity to the approach. BusMOP is an exemplar of Runtime Verification efficiency, in general applying 0% overhead to the systems it monitors. While it does only support finite state logics, it supports several, thus we feel that it can still be considered expressive.

Chapter 5 introduces the various finite state logic plugins. Of particular importance is the finite state machine plugin introduced first. The techniques presented for multicategory finite state machine minimization and enable (coenable) set gener-

ation both improve the performance of monitoring. Additionally, the technique for parallel assignments for past time linear temporal logic increases the theoretical [1] efficiency of hardware monitors derived from said logic. Lastly, linear temporal logic with past time and future time operators increases expressivity; before this thesis there was no means to monitor such formulae.

Chapter 6 introduces the context-free grammar plugin of MOP. Before the work described in that chapter, the only *efficient* parametric monitoring systems in existence used finite state based logics. The techniques of stack cloning and guaranteed acceptance allow for maintaining the semantics of "good prefix" monitoring. Additionally, a technique for deriving enable/coenable sets is presented, further improving the efficiency of parametric context-free properties. The context-free grammar plugin increases the expressivity of the MOP framework and JavaMOP in particular, without sacrificing efficiency.

Chapter 7 introduces the first parametric string rewriting monitoring algorithm. This provides the first efficient parametric monitoring algorithm for a Turing complete logic. Unfortunately, enable and coenable sets cannot be defined for string rewriting, as it is equivalent to solving the halting problem, but the results show that the plugin is still quite efficient. The string rewriting algorithm itself is the most efficient associative rewriting algorithm of which we are aware, beating associative matching in the maude [39] system by orders of magnitude. Like the context-free grammar plugin, this increases the expressivity of Runtime Verification without sacrificing too much efficiency.

Chapter 8 presents the RV-Predict system, which is able to take monitors created by JavaMOP and use them to *predict* violations properties that occur in thread interleavings other than the one experienced during a test run, showing Runtime Verification effective in yet another domain. It is also the first predictive analysis system efficient enough to operate on real world programs.

As we can see, each one of these chapters touches upon at least one of the goals of this thesis, resulting in efficient, expressive, and effective Runtime Verification.

[1] In general, bus speeds are low enough that even sequential assignments or finite state machines will always be sufficiently performant, however the technique still does produce slightly smaller monitoring circuits.

# References


[1] SPECjvm 2008. http://www.spec.org/jvm2008/.

[2] Aeronautical Radio Inc. *ARINC 653 Specification*, 2003. http://www.arinc.com/.

[3] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. 1986. pages 215–246.

[5] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable sdram memory controller. In *Hardware/software Codesign and System Synthesis (CODES+ISSS'07)*, pages 251–256, 2007.

[6] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 345–364, 2005.

[7] AspectC++. http://www.aspectc.org/.

[8] AspectJ. http://eclipse.org/aspectj/.

[9] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. ABC: an extensible AspectJ compiler. In *Aspect-Oriented System Design (AOSD'05)*, pages 87–98, 2005.

[10] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 589–608, 2007.

[11] S. Bak, D. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha. The system-level simplex architecture for improved real-time embedded system safety. In *Real-Time and Embedded Technology and Applications Symposium (RTAS'09)*, pages 99–107, 2009.

[12] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovannit-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Transactions on Computers*, 36(4):45–52, 2003.

[13] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, pages 49–69, 2004.

[14] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors. *Runtime Verification (RV'10)*, volume 6418 of *LNCS*, 2010.

[15] H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma, editors. *Runtime Verification (RV'05)*, volume 144 of *ENTCS*, 2005.

[16] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, pages 44–57, 2004.

[17] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RULER. *Journal of Logic and Computation*, pages 111–125, 2008.

[18] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass-Java with Assertions. In *Runtime Verification (RV'01)*, pages 103–117, 2001.

[19] P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior abstraction in malware analysis. In *Runtime Verification (RV'10)*, pages 168–182, 2010.

[20] P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain specific software architectures for guidance, navigation and control. *Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, 1996.

[21] Bison. `http://www.gnu.org/software/bison/`.

[22] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*, pages 169–190, 2006.

[23] E. Bodden. J-LO, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.

[24] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *Aspect-Oriented Software Development (AOSD'09)*, pages 3–14, 2009.

[25] E. Bodden and K. Havelund. Racer: effective race detection using aspectj. In *International Symposium on Software Testing and Analysis (ISSTA'08)*, pages 155–166, 2008.

[26] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming (ECOOP'07)*, pages 525–549, 2007.

[27] E. Bodden, P. Lam, and L. Hendren. Clara: a framework for statically evaluating finite-state runtime monitors. In *Runtime Verification (RV'10)*, pages 74–88, 2010.

[28] R. V. Book and F. Otto. *String-rewriting systems*. 1993.

[29] S. Chaudhuri and R. Alur. Instumenting C programs with nested word monitors. In *Model Checking Software (SPIN'07)*, pages 279–283, 2007.

[30] F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *International Conference on Formal Engineering Methods (ICFEM'04)*, pages 357–372, 2004.

[31] F. Chen, P. O. Meredith, D. Jin, and G. Roşu. Efficient formalism-independent monitoring of parametric properties. In *Automated Software Engineering (ASE'09)*, pages 383–394, 2009.

[32] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Runtime Verification (RV'03)*, pages 108–127, 2003.

[33] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 569–588, 2007.

[34] F. Chen and G. Roşu. Parametric and sliced causality. In *Computer Aided Verification (CAV'07)*, pages 240–253, 2007.

[35] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, pages 246–261, 2009.

[36] F. Chen and G. Roşu. Parametric and termination-sensitive control dependence - extended abstract. In *Static Analysis Symposium (SAS'06)*, pages 387–404, 2006.

[37] F. Chen, T. F. Şerbănuţă, and G. Roşu. jPredictor: a predictive runtime analysis tool for Java. In *International Conference on Software Engineering (ICSE'08)*, pages 221–230, 2008.

[38] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. Technical report, IBM Research, 2005.

[39] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. 2007.

[40] P. Cumming. The TI OMAP platform approach to SoCs. In *Surviving the SoC revolution: A guide to platform-based design*. 1999.

[41] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[42] M. d'Amorim and G. Roşu. Efficient monitoring of omega-languages. In *Computer Aided Verification (CAV'05)*, pages 364–378, 2005.

[43] Deflate compressed data format specification version 1.3. `http://tools.ietf.org/html/rfc1951`.

[44] D. Drusinsky. The Temporal Rover and the ATG Rover. In *Model Checking and Software Verification (SPIN'00)*, pages 323–330, 2000.

[45] A. G. Duncan. Test grammars: A method for generating program test data. In *Workshop on Software Testing and Test Documentation*, pages 270–281, 1978.

[46] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *International Conference on Software Engineering (ICSE'81)*, pages 170–178, 1981.

[47] Eagle Technology. *PCI 703 Series User's Manual*. `http://www.eagledaq.com/display_product_36.htm`.

[48] Iso/iec 14977:1996, information technology – syntactic metalanguage – extended bnf.

[49] Eiffel Language. `http://www.eiffel.com/`.

[50] P. Feiler, B. Lewis, and S. Vestal. The SAE architecture analysis & design language (AADL): A standard for engineering performance critical systems. In *Computer Aided Control Systems Design (CACSD'06)*, pages 1206–1211, 2006.

[51] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Principles of Programming Languages (POPL'04)*, pages 256–267, 2004.

[52] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. *Communications of the ACM*, 53(11):93–101, 2010.

[53] P. Gastin and D. Oddoux. Ltl with past and two-way very-weak alternating automata. In *Mathematical Foundations of Computer Science (MFCS'03)*, pages 439–448, 2003.

[54] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 385–402, 2005.

[55] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: Concepts, architectures, and implementations. *IEEE Design and Test*, 22(5):414–421, 2005.

[56] K. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[57] K. Havelund, M. Nunez, G. Roşu, and B. Wolff, editors. *Formal Approaches to Testing and Runtime Verification (FATES/RV'06)*, volume 4264 of *LNCS*, 2006.

[58] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification (RV'01)*, pages 97–114, 2001.

[59] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification (RV'01)*, pages 200–217, 2001.

[60] K. Havelund and G. Roşu, editors. *Runtime Verification (RV'02)*, volume 70 of *ENTCS*, 2002.

[61] K. Havelund and G. Roşu, editors. *Runtime Verification (RV'04)*, volume 113 of *ENTCS*, 2004.

[62] K. Havelund and G. Roşu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, pages 342–356, 2002.

[63] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Journal of Software Tools for Technology Transfer*, 6(2):158–173, 2004.

[64] C. Hoare. *Communicating Sequential Processes*. 1985.

[65] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, 1971.

[66] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, $2^{nd}$ edition*. 2001.

[67] S. Horwitz and T. W. Reps. The use of program dependence graphs in software engineering. In *International Conference on Software Engineering (ICSE'92)*, pages 392–411, 1992.

[68] B. Houssais. Verification of an algol 68 implementation. In *Strathclyde Algol 68 Conference*, pages 117–128, 1977.

[69] K. Hoyme and K. Driscoll. Safebus(tm). *IEEE Aerospace Electronics and Systems Magazine*, pages 34–39, 1993.

[70] D. Huffman. A method for the construction of minimum-redundancy codes. *Resonance*, 11:91–99, 2006.

[71] G. Hughes and T. Bultan. Interface grammars for modular software model checking. In *International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 39–49, 2007.

[72] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the Ocarina AADL tool suite. *ACM Transactions on Embedded Computing Systems*, 7(4):1–25, 2008.

[73] IBM. *Processor Local Bus Specification*, 2007. `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3BBB27E5BCC165BA87256A2B0064FFB4`.

[74] JavaMOP. `http://javamop.com`.

[75] JBoss. `http://www.jboss.org`.

[76] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu. Garbage collection for monitoring parametric properties. In *Programming Language Design and Implementation (PLDI'11)*, pages 415–424, 2011.

[77] M. Karaorman and P. Abercrombie. jcontractor: Introducing design-by-contract to java using reflective bytecode instrumentation. *Formal Methods in System Design*, 27(3):275–312, 2005.

[78] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[79] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP'01)*, pages 327–353, 2001.

[80] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, 1997.

[81] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Europoean Conference on Real-Time Systems (ECRTS'99)*, pages 114–122, 1999.

[82] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Journal of Formal Methods in System Design*, 24(2):129–155, 2004.

[83] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[84] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[85] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'00)*, pages 105–106, 2000.

[86] C. Lee, D. Jin, P. O. Meredith, and G. Roşu. Towards categorizing and formalizing the JDK API. Technical Report `http://hdl.handle.net/2142/30006`, Department of Computer Science, University of Illinois at Urbana-Champaign, 2012.

[87] B. Lickly, I. Liu, S. Kim, H. Patel, S. Edwards, and E. Lee. Predictable programming on a precision timed architecture. In *Compilers, Architecture, and Synthesis from Embedded Systems (CASES'08)*, pages 137–146, 2008.

[88] H. Lu and A. Forin. The design and implementation of P2V, an architecture for zero-overhead online verification of software programs. Technical Report MSR-TR-2007–99, Microsoft Research, 2007.

[89] M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 365–383, 2005.

[90] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Transactions on Software*, 7(4):50–55, 1990.

[91] P. O. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. In *Automated Software Engineering (ASE '08)*, pages 148–157, 2008.

[92] P. O. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *Journal of Automated Software Engineering*, pages 149–180, 2010.

[93] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *Journal on Software Techniques for Technology Transfer*, pages 249–289, 2011.

[94] P. O. Meredith and G. Roşu. Efficient parametric runtime verification with deterministic string rewriting. Technical Report `http://www.ideals.illinois.edu/handle/2142/30467`, Department of Computer Science, University of Illinois at Urbana-Champaign, 2012.

[95] B. Meyer. *Object-Oriented Software Construction,* $2^{nd}$ *edition*. 2000.

[96] NXP Semiconductors. *Philips Nexperia Digital Video Platform*. `http://www.nxp.com`.

[97] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Principles and Practice of Parallel Programming (PPoPP'03)*, pages 167–178, 2003.

[98] PCI SIG. *Conventional PCI 3.0, PCI-X 2.0 and PCI-E 2.0 Specifications*. `http://www.pcisig.com`.

[99] R. Pellizzoni, B. D. Buy, M. Caccamo, and L. Sha. Coscheduling of real-time tasks and PCI bus transactions. Technical report, University of Illinois at Urbana-Champaign, 2008.

[100] R. Pellizzoni, P. O. Meredith, M. Caccamo, and G. Roşu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time System Symposium (RTSS'08)*, pages 481–491, 2008.

[101] R. Pellizzoni, P. O. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha. Handling mixed-criticality in soc-based real-time embedded systems. In *Embedded Software (Emsoft'09)*, pages 235–244, 2009.

[102] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS'77)*, pages 46–57, 1977.

[103] Protocol buffers. `http://code.google.com/p/protobuf/`.

[104] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 2:336–375, 1972.

[105] G. Roşu, F. Chen, and T. Ball. Synthesizing monitors for safety properties – this time with calls and returns –. In *Runtime Verification (RV'08)*, pages 51–68, 2008.

[106] A.-E. Rugina, K. Kanoun, and M. Kaaniche. The ADAPT tool: From AADL architectural models to stochastic petri nets through model transformation. In *European Dependable Computing Conference (EDCC'08)*, pages 85–90, 2008.

[107] A. Sangiovanni-Vincentelli. Quo vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[108] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[109] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information System Security*, 3(1):30–50, 2000.

[110] A. Sen and V. K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *On Principles of Distributed Systems (OPODIS'03)*, pages 171–183, 2003.

[111] K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. In *Workshop on Runtime Verification (RV'03)*, pages 162–181, 2003.

[112] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Foundations of Software Engineering (FSE'03)*, pages 337–346, 2003.

[113] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Scheduling and memory requirements analysis with AADL. In *Special Interest Group on Ada (SIGAda'05)*, volume 25, pages 1–10, 2005.

[114] E. Sirer and B. Bershad. Using production grammars in software testing. In *Domain Specific Languages (DSL'00)*, pages 1–13, 1999.

[115] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *High Performance Computing, Networking, Storage and Analysis (SC'01)*, pages 8–8, 2001.

[116] O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of AADL models. In *International Parallel and Distributed Processing Symposium (IPDPS'06)*, pages 179–179, 2006.

[117] O. Sokolsky and M. Viswanathan, editors. *Runtime Verification (RV'03)*, volume 89 of *ENTCS*, 2003.

[118] R. E. Strom and S. Yemeni. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.

[119] H. Sun, M. Hauptman, and R. Lutz. Integrating product-line fault tree analysis into AADL models. In *High Assurance Systems Engineering Symposium (HASE '07)*, pages 15–22, 2007.

[120] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[121] Tracematches Benchmarks. `http://abc.comlab.ox.ac.uk/tmahead`.

[122] Uppsala University and Aalborg Univeristy. *Uppaal a tool suite for verification of real-time systems*, 2009. `http://www.uppaal.com`.

[123] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *IBM Centre for Advanced Studies Conference (CASCON'99)*, pages 125–135, 1999.

[124] C. von Praun and T. R. Gross. Object race detection. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 70–82, 2001.

[125] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Principles and Practice of Parallel Programming (PPoPP'06)*, pages 137–146, 2006.

[126] J. G. Webster. *Cardiac Pacemakers*. 1993.

[127] Xilinx, Inc. *Virtex-4 ML455 PCI/PCI-X Development Kit User Guide*, 2005. `http://www.xilinx.com/support/documentation/boards_and_kits/ug084.pdf`.

[128] Xilinx, Inc. *Virtex-5 User Guide*, 2009. `http://www.xilinx.com`.

[129] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.